The FPGA Hardware Interview:

A Topic Guide with Questions and Exercises

Hardware Interview Prep

Abstract—**Interviews for FPGA and digital-design roles draw from a stable, well-defined body of knowledge—setup/hold timing, metastability and clock-domain crossing, finite-state machines, HDL semantics, FPGA architecture, and the design of common building blocks such as FIFOs, arbiters, and synchronizers—yet candidates routinely stumble on it because the questions are asked crisply and answered under pressure. This guide is a systematic preparation resource organized as a sequence of topic reviews, each followed by interview-style questions with model answers and, for design topics, coding exercises solved in both VHDL and SystemVerilog. We cover digital-logic and arithmetic fundamentals; sequential logic and state machines; static timing analysis; metastability, synchronizers, and clock-domain crossing; clocking and reset; FPGA architecture; the semantics and common pitfalls of both HDLs; a catalogue of RTL coding idioms; memory and FIFO design; pipelining and throughput; interfaces and handshake protocols; verification and testbenches; debug and timing closure; a set of classic whiteboard problems with full solutions; a rapid-fire question bank for last-minute review; guidance on behavioral and experience questions; and a concrete study plan. The aim is that a prepared candidate walks in able to answer the expected questions fluently and to reason aloud through the unexpected ones.**

Index Terms—**FPGA, digital design, job interview, timing analysis, metastability, clock-domain crossing, finite-state machine, Verilog, SystemVerilog, VHDL, FIFO, synchronizer, verification, RTL design, interview questions.**

Contents

A structured preparation guide for digital-design and FPGA hardware engineering interviews. Each section pairs a concise review of a topic an interviewer is likely to probe with a graded set of questions and answers, and, where the topic is a design skill, with coding exercises worked out in both VHDL and SYSTEMVERILOG. The material spans the phone-screen fundamentals to the senior-level system questions, and is meant to be read once for coverage and skimmed again the night before an interview.

I. How FPGA Interviews Work

THE good news about interviewing for FPGA and digital-design roles is that the body of knowledge being tested is small, stable, and well-defined. Unlike software interviews, which can range across an unbounded space of algorithms, hardware interviews return again and again to the same core: can you reason about setup and hold, do you understand metastability and clock-domain crossing, can you design a state machine and a FIFO, and do you know what your HDL actually synthesizes to. The bad news is that these questions are asked crisply and must be answered under pressure, out loud, often at a whiteboard—and a candidate who "knows" the material but has never rehearsed saying it will stumble. This guide exists to close that gap: it collects the topics, states the questions the way an interviewer states them, and gives model answers you can practise until they are fluent.

A. The Shape of the Process

A typical loop has four stages, each probing a different thing (Fig. 1). The **recruiter screen** is non-technical—dates, role, salary—and needs only that you can describe your experience clearly. The **technical phone screen** is the first real filter: 30–60 minutes, usually one or two focused topics (very often timing or CDC) plus a small coding question over a shared editor. The **on-site or virtual loop** is several back-to-back sessions, each with a different engineer and a different emphasis—one on fundamentals, one on RTL coding at a whiteboard, one on verification or system design, one behavioral. The **behavioral** round assesses how you work: how you debug, how you handle a schedule slip, how you disagree with a reviewer. Every stage rewards the same habit: *think out loud*. An interviewer is grading your reasoning, not just your final answer, and a correct answer with no explanation scores worse than a slightly wrong one accompanied by clear thinking they can correct.

B. What Is Tested, by Level

The *topics* are the same at every level; the *depth* and the *autonomy* expected differ. Table I sketches the escalation. A junior candidate must get the fundamentals exactly right and write clean RTL for a small block. A mid-level candidate is additionally expected to reason about timing closure, CDC, and verification without prompting. A senior candidate is expected to drive system-level tradeoffs—partitioning, clocking architecture, interface choice—and to have opinions, backed by experience, about why. Knowing which level you are interviewing for tells you how deep each answer should go: at junior level "use a two-flop synchronizer" may be a complete answer; at senior level the same question expects you to discuss MTBF, multi-bit crossings, and how you would verify it.

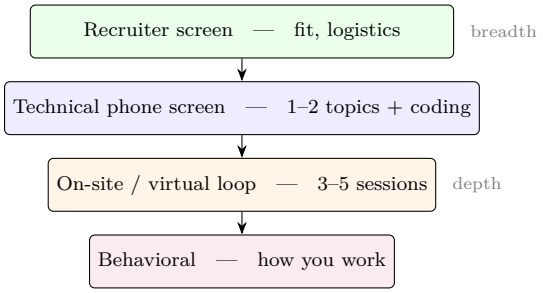

Fig. 1. The interview funnel narrows from breadth (can you talk about your work) to depth (can you design and defend real hardware). Each stage rewards reasoning aloud.

TABLE I
What each level is expected to demonstrate. Topics are shared; depth and autonomy escalate.

Level	Additionally expected to...
Junior	get fundamentals exact; write a clean small block; know setup/hold and a synchronizer.
Mid	close timing; reason about CDC and verification unprompted; design a FIFO/arbiter.
Senior	drive clocking/partitioning/interface tradeoffs; justify choices from experience; own sign-off.

C. How to Use This Guide

The document is organized as a sequence of *topic reviews*, each followed by *questions with model answers*, and—where the topic is a design skill—by *coding exercises* worked in both VHDL and SystemVerilog. Questions are marked **Q*n*** and their model answers **A**; read the question, try to answer it aloud before reading on, and only then check yourself. A worked example of the format follows immediately in the next section (§II).

The parts build in the order an interviewer's difficulty tends to build. Part II (§II–§IV) is the fundamentals any screen assumes. Part III (§V–§VII) is timing, metastability/CDC, and clocking—the single most heavily-tested area in FPGA interviews, and the one candidates most often fumble. Part IV (§VIII–§XI) covers FPGA architecture, both HDLs' semantics and pitfalls, and a catalogue of coding idioms. Part V (§XII–§XVI) is design and verification: memory/FIFOs, pipelining, interfaces, testbenches, and debug. Part VI (§XVII–§XX) is the interview itself—a set of classic whiteboard problems with full solutions, a rapid-fire question bank for the night before, guidance on behavioral questions, and a concrete study plan.

D. The Single Most Important Preparation

If you take one thing from this introduction, take this: **be able to explain setup/hold timing and metastability from first principles, out loud, in under two minutes each**. These two topics appear in essentially every FPGA interview, they are where interviewers separate candidates who memorized from candidates who understand, and they underpin half the

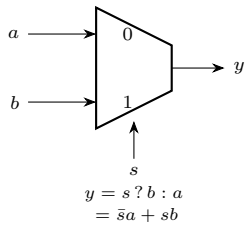

Fig. 2. A 2:1 multiplexer, the workhorse of both datapaths and the "select between two values" idiom. Every conditional in RTL becomes a mux.

other questions (CDC, clocking, FIFOs, pipelining all reduce to them). Sections V and VI treat them in depth and are the two sections to rehearse until they are automatic. Everything else in this guide is important; those two are non-negotiable.

II. Digital Logic Essentials

This section establishes the format used throughout—a short review, then questions with model answers—on the most basic material. An interviewer rarely spends long here, but they *will* ask one or two of these to calibrate, and getting a fundamental wrong is disproportionately damaging because it is supposed to be free. Answer these crisply and move on.

A. Review

Digital logic divides into **combinational** logic, whose outputs depend only on the present inputs (gates, multiplexers, decoders, adders), and **sequential** logic, whose outputs depend on inputs *and* stored state (latches, flip-flops, and everything built from them). The flip-flop is the atom of synchronous design: it samples its input on a clock edge and holds that value until the next edge, which is what lets us build pipelines and state machines with predictable timing. A **latch**, by contrast, is level-sensitive—transparent while its enable is asserted—and in synchronous FPGA design an *inferred* latch is almost always a bug (§IX, §X). The building blocks an interviewer expects you to know cold are the multiplexer (selects one of several inputs), the decoder and encoder (binary↔one-hot), the priority encoder (§XI), and the full adder (Fig. 2). Boolean simplification (algebra or Karnaugh maps) still appears, though modern synthesis makes it less central than it once was; you should be able to minimize a small function and recognize common identities such as $a + \bar{a}b = a + b$.

B. Questions

Q1. *What is the difference between a latch and a flip-flop, and why do we avoid latches in FPGA design?*

A. A flip-flop is *edge*-triggered: it samples its data input only at the active clock edge and is otherwise opaque. A latch is *level*-sensitive: it is transparent (output follows input) whenever its enable is high and holds when the enable is low. We avoid latches in synchronous FPGA design for three reasons. First, latches are usually *inferred by accident*—from an incompletely specified combinational block (a missing `else` or a `case` without `default`), so their presence signals a coding bug. Second, they complicate static timing analysis, which is built around edge-triggered flip-flops and simple clock relationships; time-borrowing through transparent latches makes timing much harder to reason about and close. Third, FPGA fabric is optimized for flip-flops (every logic cell has one), so latches map poorly. The fix is to fully specify combinational logic so a flip-flop or pure combinational function is inferred instead (§IX).

Q2. *Sketch how you would build any combinational function of n inputs using only multiplexers.*

A. A 2^n:1 multiplexer with the n inputs as its select lines implements *any* function of those inputs: wire each data input of the mux to the constant (0 or 1) that the truth table specifies for that input combination. More practically, Shannon expansion, $f = \bar{x}\, f|_{x=0} + x\, f|_{x=1}$, lets you decompose a function recursively into 2:1 muxes selected by one variable at a time—which is precisely how an FPGA lookup table (a small mux tree over a truth-table memory) realizes logic (§VIII).

Q3. *Minimize $f = \bar{a}\bar{b} + \bar{a}b + a\bar{b}$.*

A. Group terms: $\bar{a}\bar{b} + \bar{a}b = \bar{a}(\bar{b} + b) = \bar{a}$, and $\bar{a} + a\bar{b} = \bar{a} + \bar{b}$ (by $a + \bar{a}b = a + b$ applied to the complement). So $f = \bar{a} + \bar{b} = \overline{ab}$—a single NAND gate. The point the interviewer is checking is whether you know the absorption identity $a + \bar{a}b = a + b$; reaching for a Karnaugh map for three variables is fine too, and gives the same answer.

Q4. *What is the difference between a decoder and a multiplexer, and where does each appear in RTL?*

A. A decoder turns an n-bit binary code into a one-hot (2^n-bit) signal—one output asserted, selected by the input; it appears wherever you address something (selecting a register in a file, a word in a memory, a slave on a bus). A multiplexer does the reverse routing on data: it selects one of several data inputs onto a single output under a select code; it appears wherever a value is chosen conditionally. In RTL you rarely instantiate either by name—an `if/case` that assigns a signal infers a mux, and an address-comparison that enables one of several targets infers a decoder—but recognizing which structure your code implies is what lets you predict its area and timing.

Q5. *Why is one-hot encoding sometimes preferred over binary encoding for a signal or a state register on an FPGA?*

A. One-hot uses one flip-flop per state/value and decodes with a single bit test (no combinational decode of a binary code), which shortens the logic depth of next-state and output logic and therefore often improves $F_{\max}$ (§V). Flip-flops are abundant on FPGAs, so the extra registers are usually free, making the area cost negligible. Binary encoding uses fewer flip-flops but needs a decoder, adding logic levels. The rule of thumb: prefer one-hot for FSMs on FPGAs unless the state count is large; we return to state encoding in §IV.

The remaining sections follow this pattern at increasing depth. Number representation and arithmetic come next.

III. Number Representation and Arithmetic

Arithmetic questions test whether you can reason about bits precisely. They are popular because the answers are unambiguous—you either know how two's complement overflow is detected or you do not—and because fixed-point arithmetic is the daily reality of FPGA datapaths, where floating point is expensive and usually avoidable.

A. Review

Two's complement is the universal signed representation. An n-bit two's-complement number covers -2^{n-1} to $2^{n-1} - 1$: one more negative value than positive, because zero occupies a positive-side code. To negate a value, invert all bits and add one; the most-significant bit carries the sign weight -2^{n-1}, which is

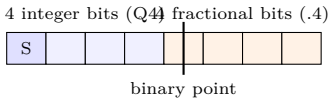

Fig. 3. A `Q4.4` fixed-point word: the MSB (S) carries the sign, four integer bits and four fractional bits straddle an implied binary point. The value is the two's-complement integer times 2^{-4}; resolution is $2^{-4} = 0.0625$.

why the arithmetic "just works" with the same adder as unsigned. **Sign extension**—replicating the MSB when widening a signed value—preserves the value; for unsigned you zero-extend instead.

Overflow is detected differently for the two interpretations. For *unsigned* addition, overflow is simply the carry-out of the top bit. For *signed* addition, overflow occurs when the carry *into* the MSB differs from the carry *out* of it—equivalently, when two operands of the same sign produce a result of the opposite sign. Adding numbers of opposite sign can never overflow.

Fixed-point represents fractional values as scaled integers. A `Q`m`.`n number has m integer bits and n fractional bits (Fig. 3); its resolution is 2^{-n}, and its signed range is -2^{m-1} to $2^{m-1} - 2^{-n}$. Two rules govern fixed-point datapaths and are frequent interview fodder: to *add*, the operands must share the same n (align the binary points first, sign-extending the integer side), and the sum grows by one bit; to *multiply*, the product's width is the sum of the operand widths and its fractional bits are the sum of the operands' fractional bits, so you must re-align (shift) and re-round afterward. On the fabric, small adders map to the dedicated fast **carry chain** and multipliers map to **DSP** slices (§VIII).

B. Questions

Q1. *Write -13 in 8-bit two's complement, and state the rule for negating any two's-complement number.*

A. Start from $+13$ = `0000_1101`. Invert to `1111_0010` and add one: `1111_0011`. So -13 = `1111_0011`. The general rule is exactly that procedure—invert all bits and add one—and it is its own inverse, so applying it to -13 returns $+13$. A useful sanity check: the value equals $-2^7 \cdot b_7 + \sum_{i<7} 2^i b_i = -128 + 64 + 32 + 16 + 2 + 1 = -13$.

Q2. *You have a signed 8-bit value and need it in 16 bits. What do you do, and how would it differ for an unsigned value?*

A. Sign-extend: replicate the MSB (bit 7) into bits 15..8. `1111_0011` (=-13) becomes `1111_1111_1111_0011`, still -13. For an *unsigned* value you zero-extend instead (bits 15..8 all zero), because there is no sign bit to preserve. In VHDL this is `resize` on `signed`/`unsigned`; in SystemVerilog, assigning a `signed` value to a wider `signed` variable sign-extends automatically, whereas an unsigned context zero-extends—getting the signedness of the operands right is the whole game.

Q3. *How does an adder detect signed overflow? How about unsigned?*

A. For unsigned addition, overflow is the carry-out of the most-significant bit. For signed (two's-complement) addition, overflow occurs when the carry into the MSB differs from the carry out of the MSB: $V = c_n \oplus c_{n-1}$. Equivalently, and easier to state in an interview: signed overflow happens only when both operands have the *same* sign and the result has the *opposite* sign—positive plus positive giving negative, or negative plus negative giving positive. Adding two operands of different signs can never overflow.

Q4. *What is the range and resolution of a `Q4.12` signed fixed-point number?*

A. It has 16 bits total: 4 integer bits (one of them the sign) and 12 fractional bits. Resolution is $2^{-12} \approx 244 \times 10^{-6}$. The signed range is -2^3 to $2^3 - 2^{-12}$, i.e. -8.0 up to just under $+8.0$. Reading the format is the skill being tested: the "4" fixes the range (the integer part), the "12" fixes the precision (the step size).

Q5. *Why do FPGA datapaths usually use fixed-point rather than floating-point arithmetic?*

A. Fixed-point add is a single integer add on the carry chain and fixed-point multiply is one DSP slice; floating-point requires alignment, normalization, and rounding hardware that costs many more LUTs, DSPs, and pipeline stages for the same throughput (§VIII). When the dynamic range of the data is known and bounded—which it usually is in signal-processing and control datapaths—fixed-point delivers the needed precision at a fraction of the area and with easier, shallower timing. Floating point earns its cost only when the dynamic range is genuinely large or unknown.

Q6. *You must add a `Q8.8` value to a `Q4.12` value. What has to happen first, and how wide is the result?*

A. The binary points must line up before adding, so you align to the finer resolution: bring both to 12 fractional bits by left-shifting the `Q8.8` operand by 4 (appending four zero LSBs), and sign-extend the integer sides to a common integer width of 8 bits. Now both are `Q8.12`; the sum grows by one integer bit to `Q9.12` to hold a possible carry. In short: align fractional bits by shifting, sign-extend integer bits, and add one bit for the sum's growth.

Q7. *After a multiply or a right-shift you must reduce precision. What is the difference between truncation and rounding, and why might you care?*

A. Truncation simply drops the low bits, which biases the result toward negative infinity (for two's complement) by up to one LSB—a systematic DC offset that accumulates in feedback loops and filters. Rounding adds half an LSB before truncating, removing most of the bias at the cost of one adder; round-half-to-even ("banker's rounding") removes the residual bias from the exact-half case. In a one-shot datapath truncation is often fine; in an accumulator, an integrator, or an IIR filter the truncation bias can drift the output, so you round. Interviewers like this question because it separates candidates who think about numerical behavior from those who only think about bits.

C. Exercise: A Saturating Signed Adder

A very common follow-up is "code an adder that saturates instead of wrapping"—the datapath equivalent of clamping. The idiom is to compute the exact sum in one extra bit (so it cannot itself overflow), then detect that the W-bit result overflowed by comparing the top two bits of the extended sum: if they differ, clamp to the extreme of the true sign. Listings 1 and 2 give it in both languages; note how the overflow test is exactly the $c_n \oplus c_{n-1}$ rule from the questions above.

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity sat_add is
  generic (W : natural := 16);
  port (clk, rst_n : in  std_logic;
        a, b       : in  signed(W-1 downto 0);
        y          : out signed(W-1 downto 0));
end entity;
architecture rtl of sat_add is
  constant MAXV : signed(W-1 downto 0) := (W-1 => '0', others => '1'); --
      011..1
```

```vhdl
  constant MINV : signed(W-1 downto 0) := (W-1 => '1', others => '0'); --
      100..0
begin
  process (clk)
    variable ext : signed(W downto 0);           -- one guard bit
  begin
    if rising_edge(clk) then
      if rst_n = '0' then
        y <= (others => '0');
      else
        ext := resize(a, W+1) + resize(b, W+1);   -- exact, sign-extended
        if ext(W) /= ext(W-1) then                -- W-bit overflow
          if ext(W) = '1' then y <= MINV;         -- true sign negative ->
      clamp min
          else y <= MAXV; end if;                 -- true sign positive ->
      clamp max
        else
          y <= resize(ext, W);
        end if;
      end if;
    end if;
  end process;
end architecture;
```

Listing 1. Saturating signed adder (VHDL).

```systemverilog
module sat_add #(parameter int W = 16) (
  input  logic                clk, rst_n,
  input  logic signed [W-1:0] a, b,
  output logic signed [W-1:0] y
);
  localparam logic signed [W-1:0] MAXV = {1'b0, {(W-1){1'b1}}}; // 011..1
  localparam logic signed [W-1:0] MINV = {1'b1, {(W-1){1'b0}}}; // 100..0
  logic signed [W:0] ext;                                  // one guard bit
  always_comb ext = {a[W-1], a} + {b[W-1], b};             // exact, sign-
      extended
  always_ff @(posedge clk) begin
    if (!rst_n)                          y <= '0;
    else if (ext[W] != ext[W-1])                          // W-bit overflow
      y <= ext[W] ? MINV : MAXV;                          // clamp to true sign
    else                                 y <= ext[W-1:0];
  end
endmodule
```

Listing 2. Saturating signed adder (SYSTEMVERILOG).

The SYSTEMVERILOG version makes the "clamp to the sign of the true result" logic explicit: on overflow, `ext[W]` is the correct sign of the mathematical sum, so a 1 means the result was too negative (clamp to `MINV`) and a 0 means too positive (clamp to `MAXV`). Getting that direction right—and explaining *why*—is the point of the exercise. State encoding and machines come next.

IV. Sequential Logic and Finite-State Machines

State machines are the single most common "code this at the whiteboard" request in an FPGA interview, and the questions around them—Moore versus Mealy, encoding choices, safe recovery—probe whether you understand the timing and robustness consequences of a design, not just its function. This section reviews the concepts and works a canonical example.

A. Review

A synchronous finite-state machine is a state register plus two blocks of combinational logic: *next-state* logic, which computes the state for the following clock edge from the current state and inputs, and *output* logic. The two classic flavors differ in how the output is formed. In a **Moore** machine the output is a function of the current *state alone*; it therefore changes only on a clock edge, is glitch-free, and can be registered, but it reacts to an input one cycle later. In a **Mealy** machine the output is a function of state *and current inputs*; it can react in the same cycle, which sometimes saves a cycle of latency, but it creates a combinational path straight from the inputs to the outputs that can glitch and that couples the two modules' timing. As a rule, prefer Moore for its clean timing unless the one-cycle-earlier response actually matters.

State encoding trades flip-flops for logic depth (§II). *Binary* uses $\lceil \log_2 N \rceil$ flip-flops for N states but needs a decoder, adding logic levels. *One-hot* uses one flip-flop per state and decodes each state with a single bit test, shortening logic and usually raising $F_{\max}$ (§V); since FPGA flip-flops are plentiful this is the default for moderate state counts. *Gray* encoding, in which adjacent states differ by one bit, matters mainly when the state value itself crosses a clock domain (§VI). Modern synthesis tools will re-encode an FSM automatically unless you constrain them, but you should be able to justify a choice.

Coding style matters for correctness. The common structures are the *two-process* form (one clocked process for the state register, one combinational process computing both next-state and output) and the *three-process* form (state register, next-state, output as three separate blocks). The one discipline that is non-negotiable: the combinational next-state process must assign the next-state variable on *every* path—a default assignment at the top, or a `default`/`when others` branch—or you will infer a latch (§IX). Providing a `when others`/`default` that returns to a known state also gives *safe recovery*: if the state register is ever corrupted into an unused code (by an upset, or after reset glitches), the machine falls back to a defined state instead of hanging.

B. Questions

Q1. *What is the difference between a Moore and a Mealy machine, and when would you choose each?*

A. A Moore machine's outputs depend only on the current state, so they are synchronous with the clock, glitch-free, and easy to register; the cost is that the machine responds to an input change one cycle later. A Mealy machine's outputs depend on state and inputs together, so it can respond in the same cycle and sometimes needs fewer states, but its outputs can glitch and there is a direct combinational path from inputs to outputs that entangles the timing of whatever drives the inputs and whatever the outputs feed. Choose Moore by default for clean, predictable timing; choose Mealy when the one-cycle-earlier response is genuinely needed or when it collapses the state count significantly, and register the Mealy output if the downstream logic needs it glitch-free.

Q2. *On an FPGA, when would you pick one-hot encoding over binary for the state register?*

A. Almost always, for small-to-moderate state counts. One-hot uses one flip-flop per state—cheap, since FPGAs have a flip-flop in every logic cell—and lets next-state and output logic test a single bit instead of decoding a binary code, which reduces logic depth and tends to improve $F_{\max}$. Binary encoding wins only when the state count is large enough that one-hot's flip-flop or routing cost becomes significant, or when you need the compact encoded value for something else. If the state value crosses a clock domain, neither is right—you want gray or a handshake (§VI).

Q3. *How do you make an FSM robust against ending up in an illegal state?*

A. Two measures. First, give the next-state logic a `when others` (VHDL) or `default` (SystemVerilog) branch that steers any unused encoding back to the reset/idle state, so an illegal code cannot hang the machine. Second, for designs that care (radiation, safety), enable the tool's "safe FSM" synthesis option, which adds explicit recovery logic for all unreachable codes—something one-hot needs in particular, because most of its 2^N codes are illegal. And of course reset the state register to a defined initial state. Interviewers ask this to see whether you think about failure modes, not just the happy path.

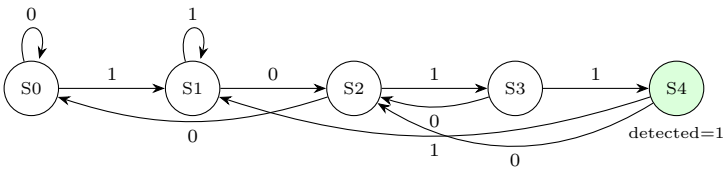

Fig. 4. Moore state diagram for an overlapping `1011` detector. Each state is the longest matched prefix; `S4` asserts the output. The edges out of `S4` (`1`→`S1`, `0`→`S2`) implement overlap.

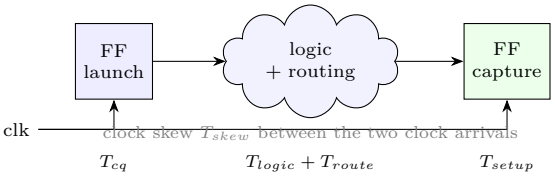

Fig. 5. A register-to-register timing path: launch flop, combinational logic and routing, capture flop. STA sums the delays and checks them against the clock period (setup) and against zero (hold).

Q4. *Why might you register the outputs of a state machine?*
A. A registered (Moore, flip-flopped) output is glitch-free and has a clean, short output path—it comes straight from a flip-flop—so it meets timing at the module boundary and will not produce transient pulses that could, for example, falsely trigger a downstream edge-detector or an asynchronous input. Unregistered combinational outputs (especially Mealy) can glitch as the next-state logic settles and present a long combinational path to the consumer. The cost of registering is one cycle of latency, which is usually acceptable.

Q5. *What is the danger in the combinational next-state process of a two-process FSM, and how do you avoid it?*
A. If any path through the combinational process fails to assign the next-state variable, the tool infers a latch to "remember" the unassigned value—an accidental, level-sensitive element that breaks the synchronous timing model and is a classic bug (§IX). Avoid it by assigning a default value to the next-state variable at the top of the process (typically "stay in the current state") before the `case`, and by including a `when others/default` branch. The same discipline makes every combinational process safe, not just FSMs.

C. Exercise: A "1011" Sequence Detector

The canonical FSM exercise is an overlapping serial sequence detector. We build a Moore detector for the pattern `1011` that asserts `detected` for one cycle whenever the pattern completes, and—because it overlaps—reuses the trailing `1` as the possible start of the next match. Fig. 4 is the state diagram; each state represents the longest prefix of `1011` matched so far, and the detect state `S4` on a subsequent `1` returns to `S1` (matched "`1`") rather than to the idle state, which is exactly what makes the detector overlapping.

```vhdl
library ieee; use ieee.std_logic_1164.all;
entity seq1011 is
  port (clk, rst_n, din : in  std_logic;
        detected        : out std_logic);
end entity;
architecture rtl of seq1011 is
  type state_t is (S0, S1, S2, S3, S4);
  signal state, nstate : state_t;
begin
  -- state register
  process (clk) begin
    if rising_edge(clk) then
      if rst_n = '0' then state <= S0; else state <= nstate; end if;
    end if;
  end process;
  -- next-state (combinational, fully specified)
  process (state, din) begin
    case state is
      when S0 => if din = '1' then nstate <= S1; else nstate <= S0; end if;
      when S1 => if din = '1' then nstate <= S1; else nstate <= S2; end if;
      when S2 => if din = '1' then nstate <= S3; else nstate <= S0; end if;
      when S3 => if din = '1' then nstate <= S4; else nstate <= S2; end if;
      when S4 => if din = '1' then nstate <= S1; else nstate <= S2; end if;
      when others => nstate <= S0;              -- safe recovery
    end case;
  end process;
  detected <= '1' when state = S4 else '0';     -- Moore output
end architecture;
```

Listing 3. Overlapping "1011" Moore detector (VHDL).

```systemverilog
module seq1011 (
  input  logic clk, rst_n, din,
  output logic detected
);
  typedef enum logic [2:0] {S0, S1, S2, S3, S4} state_t;
  state_t state, nstate;

  always_ff @(posedge clk)
    if (!rst_n) state <= S0; else state <= nstate;

  always_comb begin
    nstate = S0;                                // default -> safe recovery
    unique case (state)
      S0: nstate = din ? S1 : S0;
      S1: nstate = din ? S1 : S2;
      S2: nstate = din ? S3 : S0;
      S3: nstate = din ? S4 : S2;
      S4: nstate = din ? S1 : S2;               // overlap: reuse trailing 1
    endcase
  end

  assign detected = (state == S4);              // Moore output
endmodule
```

Listing 4. Overlapping "1011" Moore detector (SYSTEMVERILOG).

Two details win points here. First, the output is Moore—a pure function of `state`—so it is glitch-free and one cycle after the completing bit; if the interviewer wants same-cycle detection, that is the Mealy variant (assert on the `S3`, `din=1` *edge* rather than in `S4`), and you should be able to state the tradeoff. Second, both the `when others/default` and the default `nstate` assignment give safe recovery and prevent an inferred latch—exactly the disciplines from the questions above. Static timing analysis, the topic these machines must ultimately satisfy, is next (§V).

V. Static Timing Analysis

If you prepare only one section of this guide, prepare this one and the next. Setup/hold reasoning is asked in essentially every FPGA interview, it is the foundation of clock-domain crossing (§VI), clocking (§VII), pipelining (§XIII), and FIFO design (§XII), and it is the single most reliable way an interviewer separates candidates who memorized definitions from candidates who understand why the definitions are what they are. Rehearse the equations until you can write them from memory and explain every term aloud.

A. Review

Static timing analysis (STA) verifies, without simulation, that every register-to-register path in a design meets its timing constraints across all paths and corners. The unit of analysis is the **timing path**: it launches at a source flip-flop's clock edge, propagates through the flop's clock-to-output delay, then through combinational logic and routing, and must arrive at a destination flip-flop's data input in time for the destination clock edge (Fig. 5).

Two checks apply to every path. The **setup check** says the data must arrive and be stable at least T_{setup} *before* the capture edge; it is a *max-delay* check, and it is the one bounded by the

clock period. The **hold check** says the data must remain stable at least T_{hold} *after* the capture edge (so the new launched value does not race through and corrupt the value being captured for the *current* edge); it is a *min-delay* check, and—crucially—it does not involve the clock period at all.

The setup slack, with all terms positive, is

$$\text{slack}_{setup} = T_{clk} - T_{skew} - \big(T_{cq} + T_{logic} + T_{route} + T_{setup}\big).$$

The design meets setup when this is ≥ 0. The maximum operating frequency is $F_{\max} = 1/T_{clk,\min}$, where $T_{clk,\min}$ is the smallest period for which the worst path's setup slack is exactly zero. The hold check, using *minimum* path delay, is

$$T_{cq} + T_{logic,\min} + T_{route,\min} \geq T_{hold} + T_{skew},$$

i.e. the earliest the new data can arrive must still be later than the hold window of the capture edge.

The single most important consequence: T_{clk} appears in the setup equation but *not* in the hold equation. Increasing the period (slowing the clock) increases setup slack but leaves hold slack completely unchanged. Therefore a hold violation cannot be fixed by slowing the clock—it must be fixed by adding delay on the short path, by reducing clock skew, or (in the FPGA back-end) by the router inserting delay. This is the classic "gotcha" question and you must be able to explain *why* from the equations, not just assert it.

Clock skew is the difference in a clock edge's arrival time at the two flops. Positive ("helpful") skew—the capture clock arriving *later* than the launch clock—adds to the setup budget but subtracts from the hold budget; it helps setup and hurts hold. **Jitter** is cycle-to-cycle variation in the clock edge; together with skew and a margin it is folded into **clock uncertainty**, which the tool subtracts from the available period before checking setup (and can add for hold). **WNS** (worst negative slack) is the most negative slack over all paths; **TNS** (total negative slack) is the sum of all negative slacks—WNS tells you the worst path, TNS tells you how much total work remains to close timing. **Multi-cycle paths** (a result allowed several cycles) and **false paths** (paths that can never be exercised, or that cross unrelated clocks) are exceptions you tell the tool about so it does not waste effort or report spurious failures.

B. Questions

Q1. *Define setup time and hold time.*

A. Setup time T_{setup} is the minimum interval *before* the active clock edge during which the data input must be stable for the flip-flop to capture it reliably. Hold time T_{hold} is the minimum interval *after* the active clock edge during which the data input must remain stable. Together they define a window around the edge in which the input must not change; a change inside that window risks metastability (§VI). Setup bounds how *late* data may arrive; hold bounds how *early* the next data may arrive.

Q2. *Write the setup-slack equation and name each term.*

A. slack $= T_{clk} - T_{skew} - (T_{cq} + T_{logic} + T_{route} + T_{setup})$. Here T_{clk} is the clock period (the time available); T_{skew} is the clock-arrival difference between launch and capture flops (subtracted for a setup check when it is "harmful," i.e. capture earlier); T_{cq} is the launch flop's clock-to-Q delay; T_{logic} and T_{route} are the combinational-logic and interconnect delays along the path (on an FPGA, routing often dominates); and T_{setup} is the capture flop's setup requirement. The path meets setup when slack ≥ 0.

Q3. *Why can't you fix a hold violation by slowing down the clock?*

A. Because the clock period does not appear in the hold check. The hold constraint, $T_{cq} + T_{logic,\min} \geq T_{hold} + T_{skew}$, compares the *minimum* propagation delay of the launched data against the hold window of the *same* capture edge that just launched it—a same-edge race that is independent of how far away the next edge is. Slowing the clock moves the *next* edge later, which only affects setup. A hold violation is fixed by adding delay on the fast path, reducing clock skew, or letting the router insert delay; it is never fixed by frequency.

Q4. *What physically causes a hold violation, and why are they common on short paths?*

A. A hold violation occurs when data launched by a clock edge races through a very short combinational path and reaches the capture flop *too soon*—while that flop is still holding the value for the current edge—so it corrupts the captured value. Short paths (a direct flop-to-flop connection, or logic with tiny delay) are the culprits precisely because their propagation delay is small relative to the hold requirement plus any clock skew. Clock skew makes it worse: if the capture clock arrives later than the launch clock, the effective hold window widens. This is why hold problems are a placement/skew/routing issue, not a logic-depth issue.

Q5. *What are WNS, TNS, and $F_{\max}$?*

A. WNS (worst negative slack) is the single most negative slack across all timing paths—it identifies the critical path and whether the design meets timing at all (WNS ≥ 0 means it passes). TNS (total negative slack) is the sum of the negative slacks of all failing endpoints—a measure of how much total timing work remains, so a design with WNS $= -0.1$ ns and TNS $= -0.1$ ns is one fix away, while the same WNS with TNS $= -40$ ns has hundreds of failing paths. $F_{\max}$ is the highest clock frequency at which setup is met on the worst path: $F_{\max} = 1/(T_{clk} - \text{WNS})$ evaluated at the constrained period, i.e. the period shrunk until WNS hits zero.

Q6. *How does clock skew affect setup versus hold, and what is "helpful" skew?*

A. Skew is the arrival-time difference of a clock edge at the two flops. If the capture clock arrives *later* than the launch clock (positive skew relative to the data direction), the data has more time to arrive—this *helps setup*. But that same later capture edge also extends the window during which the next launched data must not arrive—this *hurts hold*. So skew trades setup margin for hold margin. "Helpful" (or "useful") skew is deliberately introducing skew to borrow time for a critical setup path, accepting the hold cost, which the tools can do automatically during clock-tree synthesis and placement.

Q7. *What is clock uncertainty, and where does jitter enter?*

A. Clock uncertainty is a lumped margin the timing tool subtracts from the available period before the setup check (and adds for hold) to account for non-ideal clocks: it bundles clock *jitter* (cycle-to-cycle edge variation, largely from the PLL/MMCM and supply noise), a portion of skew that is not path-specific, and any user-specified guardband. In the setup equation it effectively reduces T_{clk}; in the hold equation it effectively increases the required margin. On an FPGA you rarely compute it by hand—the tool derives it from the clock source—but you must know it exists and why a design that "should" close at exactly the target period does not.

C. Worked Numeric Exercise

Setup. A path has $T_{cq} = 0.4$ ns, combinational logic $T_{logic} = 2.1$ ns, routing $T_{route} = 1.3$ ns, capture $T_{setup} = 0.2$ ns, and clock skew $T_{skew} = 0.1$ ns (harmful). The clock period is $T_{clk} = 5.0$ ns (200 MHz).

Setup slack $= 5.0 - 0.1 - (0.4 + 2.1 + 1.3 + 0.2) = 5.0 - 0.1 - 4.0 = +0.9$ ns. The path passes with 0.9 ns to spare. The minimum period for this path is $T_{clk,\min} = T_{skew} + T_{cq} + T_{logic} + T_{route} + T_{setup} = 0.1 + 4.0 = 4.1$ ns, so $F_{\max} = 1/4.1\,\text{ns} \approx 244$ MHz for this path (the design $F_{\max}$ is set by the *worst* path over all paths).

Hold. Suppose the shortest path between two flops has $T_{cq} + T_{logic,\min} = 0.5$ ns, the capture flop needs $T_{hold} = 0.3$ ns, and clock skew is $T_{skew} = 0.1$ ns (capture later). *Hold check:* require $0.5 \geq 0.3 + 0.1 = 0.4$. Since $0.5 \geq 0.4$, hold passes with 0.1 ns of margin. Note the clock period never entered this calculation—confirming that hold is period-independent.

D. Constraining the Clock

Timing analysis needs a clock definition; the primary constraint is `create_clock`, and getting it right is a prerequisite for every check above (§VII, §XVI).

```
# 200 MHz clock on the input port clk_in (period in ns)
create_clock -name sysclk -period 5.000 [get_ports clk_in]
# fold jitter/margin into uncertainty (often auto-derived)
set_clock_uncertainty 0.150 [get_clocks sysclk]
# after implementation, look at the worst setup and hold paths
report_timing -delay_type max -max_paths 5   ;# worst setup (WNS)
report_timing -delay_type min -max_paths 5   ;# worst hold
```

Listing 5. Defining a clock and inspecting the worst path (XDC).

For a deeper, generator-level treatment of timing closure on a real core, see the companion RTL Processor Design series [14]; here the goal is only that you can derive and explain these checks under interview conditions. The next section applies them to the case that breaks them on purpose: asynchronous inputs and clock-domain crossing.

VI. Metastability, Synchronizers, and CDC

This is the other section to rehearse until it is automatic. Clock-domain crossing (CDC) bugs are the ones that pass every simulation and fail intermittently in the lab, so interviewers probe hard here to find out whether you truly understand the failure mechanism or have only memorized "add two flip-flops." Be ready to derive the reasoning, not recite the recipe.

A. Review

Metastability is what happens when a flip-flop's setup/hold window is violated—which is unavoidable when sampling a signal from an asynchronous clock domain, because the arriving edge bears no fixed relationship to the sampling edge. The flop's internal node is driven to a point between the two stable logic levels (a balance point of its cross-coupled pair) and takes an *unbounded*, statistically-distributed time to resolve to a valid 0 or 1. It always resolves eventually; the danger is that it may still be ambiguous when downstream logic samples it, and—worse—two gates reading a metastable node can resolve it to *different* values, corrupting state.

The probability of *surviving* metastability past a settling time follows an exponential law, giving the mean time between failures

$$\text{MTBF} = \frac{e^{\,t_r/\tau}}{f_c\, f_d\, T_0},$$

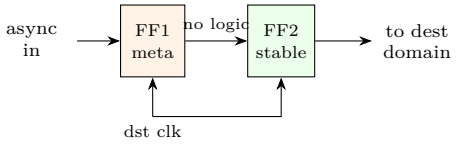

Fig. 6. The two-flop synchronizer for a single bit. FF1 may go metastable but drives only FF2, which samples it a full clock later, granting the settling time t_r that dominates MTBF.

where t_r is the settling time allowed before the signal is used, τ is the flop's metastability resolution time constant (a device/process parameter), T_0 is a device parameter for the width of the metastability-susceptible window, f_c is the capturing clock frequency, and f_d is the toggle rate of the asynchronous data. The denominator counts how often you *enter* metastability; the exponential numerator is how reliably you *escape* it in the time allowed. Because t_r sits in the exponent, buying more settling time pays off enormously.

That is exactly what a **two-flip-flop synchronizer** does: the first flop may go metastable, but nothing reads its output; a second flop, one clock later, gives the first flop nearly a full clock period ($t_r \approx T_{clk}$) to resolve before anyone downstream sees the value (Fig. 6). One extra period of t_r in the exponent turns an MTBF of microseconds into one of millions of years. Two rules make it work: (1) *no combinational logic* between the first and second stage—nothing may sample the possibly-metastable first-stage output before the second flop does; and (2) the flops must be placed tightly together (in FPGA tools, the `ASYNC_REG` attribute) so routing delay does not eat the settling budget.

Single bit versus multi-bit is the trap. A two-flop synchronizer is correct only for a *single* bit (a level, or a slow-changing flag). You *cannot* put one on each bit of a multi-bit bus, because the individual bits go metastable and resolve *independently*: on a cycle where several bits change, some may resolve to their new value and some to their old, yielding a sampled word that is neither the old nor the new value—a value that never existed. The safe multi-bit techniques are:

- **Gray code** for counters/pointers—consecutive values differ in exactly one bit, so any sample is either the old or the new value, never a mix. This is why asynchronous FIFOs cross their pointers in Gray code.
- **Req/ack handshake** for arbitrary data—the source holds the data stable and asserts a request (itself synchronized); the destination captures the stable data and acknowledges. Robust but low throughput.
- **Asynchronous FIFO** for streaming data—a dual-clock FIFO whose read/write pointers cross as Gray codes; the workhorse for moving data volumes between domains.

A related hazard is crossing a **pulse**: a single-cycle pulse in a fast domain may be missed entirely by a slower destination, or stretched. The **toggle synchronizer** converts the pulse into a level toggle in the source, synchronizes the level, and edge-detects it in the destination to regenerate a pulse. Finally, every crossing must be *constrained* so STA does not try to time it as if it were synchronous—typically `set_false_path` on the crossing, or the more conservative `set_max_delay -datapath_only` which still bounds the routing delay into the first flop.

B. Questions

Q1. *What is metastability, and when does it occur?*

A. Metastability is a flip-flop entering a temporarily undefined state—its output hovering between valid logic levels—after its setup or hold time is violated. It occurs whenever a flop samples a signal that can change within its setup/hold window, which is inherent when sampling data from an asynchronous clock domain (the source edge has no fixed phase relationship to the sampling edge). The flop resolves to a valid level after an unbounded, random time; the risk is that downstream logic samples it before it has resolved.

Q2. *Quantitatively, why does adding a second flip-flop help?*

A. Because the settling time sits in the exponent of the MTBF law, MTBF $= e^{t_r/\tau}/(f_c f_d T_0)$. A single sampling flop leaves almost no settling time before its value is used, so $t_r \approx 0$ and MTBF is tiny. Inserting a second flop clocked one period later means nothing reads the first flop's output for nearly a full clock period, so $t_r \approx T_{clk}$. Since MTBF grows exponentially in t_r/τ, that one extra period typically lifts MTBF from microseconds to geological timescales. The second flop buys time, and time is exponentially valuable.

Q3. *Can you synchronize an 8-bit bus by putting a two-flop synchronizer on each bit? Why or why not?*

A. No. Each bit's synchronizer resolves metastability *independently*, so on a cycle where several bits are changing, some bits may be captured at their new value and others at their old value in the same destination cycle. The result is a word that is neither the pre-change nor the post-change value—a transient that never existed in the source. For a binary counter crossing, this can be a wildly wrong count. Multi-bit values must cross via Gray coding, a handshake, or an asynchronous FIFO—never per-bit two-flop synchronizers.

Q4. *How do you safely cross a multi-bit value between clock domains?*

A. It depends on the value: (1) a *monotonic counter or pointer* crosses as a *Gray code*, because only one bit changes per increment, so any sample is old-or-new but never a mix; (2) an *arbitrary data word* that changes occasionally crosses via a *req/ack handshake*—hold the data stable, synchronize a request, capture on the destination side, synchronize an acknowledge back; (3) a *stream* of data crosses through an *asynchronous FIFO*, which internally uses Gray-coded pointers and a dual-clock memory (§XII). Match the mechanism to the traffic: handshake for control, FIFO for bandwidth.

Q5. *Why do asynchronous FIFOs use Gray-coded pointers?*

A. Because the read and write pointers must cross into the opposite clock domain to compute the full and empty flags, and a pointer is a multi-bit value. In Gray code, incrementing changes exactly one bit, so when the opposite domain samples the pointer mid-transition it gets either the pre-increment or the post-increment value—never a spurious intermediate. That guarantees the full/empty comparisons are conservative (they may lag by one, which is safe, but never see a corrupt pointer that could cause overflow or underflow).

Q6. *What is the rule about logic on the first synchronizer flop, and why?*

A. No combinational logic may be placed on the output of the first synchronizer flop before the second flop samples it (and nothing else may fan out from it). The first flop's output can be metastable; if logic reads it during that window, different gates may interpret the ambiguous level differently and the settling budget is spent on logic delay instead of on resolution. The first-stage output must go *only* to the second flop's input, and the pair should be placed tightly (`ASYNC_REG`) so routing delay does not steal settling time.

Q7. *How do you constrain a CDC path for timing?*

A. You tell STA not to time it as a synchronous path. The common options are `set_false_path` on the path into the first synchronizer flop (removes the setup/hold check entirely between the unrelated clocks), or the more conservative `set_max_delay -datapath_only`, which ignores the clock relationship but still bounds the routing delay so the two synchronizer flops stay close. For whole unrelated clock domains, `set_clock_groups -asynchronous` declares them unrelated in one statement. Constraining the crossing is mandatory—an unconstrained CDC path either reports as a false failure or, worse, gets "optimized" in a way that breaks the synchronizer.

C. Coding Exercise: Synchronizers

Interviewers often ask you to write a synchronizer on the spot. Listings 6 and 7 give the two-flop single-bit synchronizer in both languages, and Listing 8 gives the toggle (pulse) synchronizer. Note the `ASYNC_REG` attribute in each—mentioning it unprompted signals real experience.

```vhdl
library ieee; use ieee.std_logic_1164.all;
entity sync2 is
  port (dst_clk, rst_n, async_in : in  std_logic;
        sync_out                 : out std_logic);
  attribute ASYNC_REG : string;
end entity;
architecture rtl of sync2 is
  signal meta_q, sync_q : std_logic := '0';
  attribute ASYNC_REG of meta_q : signal is "TRUE";
  attribute ASYNC_REG of sync_q : signal is "TRUE";
begin
  process (dst_clk) begin
    if rising_edge(dst_clk) then
      if rst_n = '0' then meta_q <= '0'; sync_q <= '0';
      else
        meta_q <= async_in;  -- may go metastable
        sync_q <= meta_q;    -- only reader of meta_q
      end if;
    end if;
  end process;
  sync_out <= sync_q;
end architecture;
```

Listing 6. Two-flop single-bit synchronizer (VHDL).

```systemverilog
module sync2 (input logic dst_clk, rst_n, async_in,
              output logic sync_out);
  (* ASYNC_REG = "TRUE" *) logic meta_q, sync_q;
  always_ff @(posedge dst_clk)
    if (!rst_n) begin meta_q <= 1'b0; sync_q <= 1'b0; end
    else        begin meta_q <= async_in; sync_q <= meta_q; end
  assign sync_out = sync_q;
endmodule
```

Listing 7. Two-flop single-bit synchronizer (SYSTEMVERILOG).

```systemverilog
// Source domain: turn each src_pulse into a level toggle.
always_ff @(posedge src_clk)
  if (!rst_n)         tog <= 1'b0;
  else if (src_pulse) tog <= ~tog;

// Destination: 2-flop sync the level, then edge-detect it.
(* ASYNC_REG = "TRUE" *) logic s0, s1; logic s2;
always_ff @(posedge dst_clk) begin
  {s2, s1, s0} <= {s1, s0, tog};        // 2-FF sync + one delay
end
assign dst_pulse = s1 ^ s2;             // one pulse per source event
```

Listing 8. Toggle (pulse) synchronizer: a source pulse becomes one destination pulse regardless of the frequency ratio (SYSTEMVERILOG).

CDC is a large subject; the companion RTL Processor Design series devotes an entire volume to it [14], and Cummings' notes [3], [4] are the standard references. For the interview, the essentials above—the failure mechanism, the MTBF exponent, single- versus multi-bit, and the ability to write a synchronizer

from memory—cover the overwhelming majority of what is asked.

VII. Clocking and Reset

Clocking and reset questions probe whether you understand that on an FPGA the clock and reset networks are *special*—dedicated resources with rules that differ from ordinary logic—and whether you have internalized the two idioms that follow from those rules: use clock *enables* rather than gated clocks, and reset with *asynchronous assertion, synchronous deassertion*. Get these two right and you have answered most of what an interviewer will ask here.

A. Review

Clocks ride dedicated networks. A clock must reach thousands of flip-flops with minimal skew, so FPGAs route clocks on dedicated global buffers (`BUFG`) and low-skew spines that fan out across *clock regions*, not on ordinary logic routing. A signal you generate in the fabric and try to use as a clock (a "ripple clock" from a counter bit, a divided clock built in LUTs) does not get this network, arrives skewed, and is a classic red flag.

Enable, don't gate. To stop a register from updating, you do not gate its clock in the fabric—a combinational gate on a clock line glitches and destroys the low-skew guarantee. Instead you use the flip-flop's *clock enable* (CE): the clock runs freely and the register simply ignores edges when CE is low. Every FPGA flip-flop has a CE input, so this is free. For coarse, whole-domain gating (to save power on an idle block) there is a dedicated glitch-free clock-enable buffer, `BUFGCE`, which cleanly starts and stops a global clock at the buffer, not in the fabric.

Clock generation. A `PLL` or `MMCM` synthesizes new clock frequencies from a reference (multiplying and dividing), removes/aligns phase, and can de-skew; it asserts a `locked` signal when its output is stable, which should hold the design in reset until it is high.

Reset discipline. A purely *synchronous* reset is sampled like any other input and is easy to time, but needs a running clock to take effect. A purely *asynchronous* reset acts immediately regardless of the clock, but its *release* is the danger: if reset deasserts too close to a clock edge, the flip-flop can go metastable on the release, exactly as a data input would (this is checked by *recovery* and *removal* times—the async-control analogues of setup and hold). The standard fix is the *reset synchronizer* (reset bridge): assert reset asynchronously (so it works even with no clock) but deassert it *synchronously*, by passing the deassertion through two flip-flops clocked by the destination domain. On FPGAs many datapath registers also power up to a known state from the bitstream (global set/reset), so they need no explicit reset at all—reserve explicit reset for control state that must be initialized, which reduces reset fanout and helps timing.

B. Questions

Q1. *Why should you not gate a clock in FPGA fabric to save power, and what do you do instead?*

A. A combinational gate placed on a clock net can glitch (produce a spurious edge) as its inputs settle, and it adds skew that the dedicated clock network was built to avoid—both can cause unintended captures or timing failures. Instead you use the flip-flop's *clock enable*: the clock free-runs and the register updates only on cycles when CE is asserted, which is glitch-free and free in area because every FPGA flip-flop has a CE. For coarse power gating of a whole clock domain, use a dedicated `BUFGCE`, which enables/disables the global clock at the buffer without introducing a fabric gate.

Q2. *Compare synchronous and asynchronous reset. Which do you recommend on an FPGA, and how do you implement it?*

A. A synchronous reset is sampled on the clock edge—simple to time and it filters glitches, but it requires a running clock to take effect and adds to the data-path logic. An asynchronous reset takes effect immediately without a clock, which is useful at power-up, but its *deassertion* can violate recovery/ removal and drive the flip-flop metastable. The recommended scheme is **asynchronous assertion, synchronous deassertion**: assert reset immediately (works with no clock), but release it through a two-flop synchronizer in the destination clock domain so the deasserting edge is clean. Implement one such reset synchronizer per clock domain and distribute its output.

Q3. *What are recovery and removal times?*

A. They are the timing constraints on an *asynchronous* control input (such as an async reset or preset) relative to the clock edge, analogous to setup and hold on a data input. *Recovery* time is the minimum time the async signal must be *deasserted before* the active clock edge; *removal* time is the minimum time it must remain asserted *after* the edge. Violating them on reset release is precisely why we synchronize reset deassertion.

Q4. *What does an MMCM or PLL do, and why hold the design in reset until it locks?*

A. It synthesizes one or more output clocks from an input reference by multiplying and dividing the frequency and adjusting phase, and it can reduce clock skew/jitter. While it is acquiring lock, its output frequency and phase are unstable, so any logic clocked by it would behave unpredictably; the `locked` output goes high once the clock is stable, and the design should remain in reset until then.

Q5. *How do you get a reset from one clock domain safely into another?*

A. You do not route a single reset flip-flop's output across domains. Each clock domain gets its own reset synchronizer: the incoming (possibly asynchronous) reset asserts all of them immediately, but each deasserts synchronously to its own clock through its own two-flop bridge. This is the reset case of clock-domain crossing (§VI).

C. Exercise: A Stallable Register and a Reset Synchronizer

Two building blocks capture this section. First, a pipeline register with a clock enable (`en`)—the "enable, don't gate" idiom for stalling a pipeline stage. Second, the reset synchronizer: asynchronous assert, synchronous deassert. Note the `ASYNC_REG` attribute on the synchronizer flops so the tools place them close for maximum metastability settling (§VI).

```
library ieee; use ieee.std_logic_1164.all;

-- (1) Stallable pipeline register: enable, don't gate the clock.
entity ce_reg is
  generic (WIDTH : natural := 8);
  port (clk, en : in std_logic;
        d : in  std_logic_vector(WIDTH-1 downto 0);
        q : out std_logic_vector(WIDTH-1 downto 0));
end entity;
architecture rtl of ce_reg is begin
  process (clk) begin
    if rising_edge(clk) then
      if en = '1' then q <= d; end if;   -- clock free-runs; CE gates update
    end if;
  end process;
```

```vhdl
end architecture;

-- (2) Reset synchronizer: async assert, sync deassert.
library ieee; use ieee.std_logic_1164.all;
entity reset_sync is
  port (clk       : in  std_logic;
        arst_n    : in  std_logic;       -- async, active-low reset in
        rst_n_out : out std_logic);      -- sync-deasserted reset out
end entity;
architecture rtl of reset_sync is
  signal s : std_logic_vector(1 downto 0) := "00";
  attribute ASYNC_REG : string;
  attribute ASYNC_REG of s : signal is "TRUE";
begin
  process (clk, arst_n) begin
    if arst_n = '0' then                 -- ASYNCHRONOUS assertion
      s <= "00";
    elsif rising_edge(clk) then          -- SYNCHRONOUS deassertion
      s <= s(0) & '1';
    end if;
  end process;
  rst_n_out <= s(1);
end architecture;
```

Listing 9. Clock-enable register and async-assert/sync-deassert reset synchronizer (VHDL).

```systemverilog
// (1) Stallable pipeline register.
module ce_reg #(parameter int WIDTH = 8) (
  input  logic             clk, en,
  input  logic [WIDTH-1:0] d,
  output logic [WIDTH-1:0] q
);
  always_ff @(posedge clk)
    if (en) q <= d;              // clock free-runs; CE gates the update
endmodule

// (2) Reset synchronizer: async assert, sync deassert.
module reset_sync (
  input  logic clk,
  input  logic arst_n,          // async, active-low reset in
  output logic rst_n_out        // sync-deasserted reset out
);
  (* ASYNC_REG = "TRUE" *) logic [1:0] s;
  always_ff @(posedge clk or negedge arst_n)
    if (!arst_n) s <= 2'b00;    // ASYNCHRONOUS assertion
    else         s <= {s[0], 1'b1};  // SYNCHRONOUS deassertion
  assign rst_n_out = s[1];
endmodule
```

Listing 10. The same two blocks (SYSTEMVERILOG).

The reset synchronizer is worth being able to draw and code from memory: it is one of the two or three blocks an interviewer may ask you to write on the spot, and it directly exercises the metastability reasoning of §VI. For a fuller treatment of FPGA clocking and reset architecture see the RTL Processor Design series [14] and the vendor methodology guide [3], [10].

VIII. FPGA Architecture

Architecture questions check that you know what your RTL actually becomes on the die. An FPGA is not a blank slate; it is a fixed grid of specific primitives—lookup tables, flip-flops, block RAMs, DSP slices, carry chains—and good FPGA engineers write RTL with those primitives in mind. Interviewers ask this to see whether you will, for example, reach for a DSP when you multiply, a block RAM when you need a kilobyte, and a register when you need a byte.

A. Review

The LUT is the unit of logic. A k-input lookup table is a 2^k-bit truth-table memory read out by a mux tree addressed by the k inputs. Because you can load any truth table into it, one LUT implements *any* Boolean function of its k inputs; modern devices use $k = 6$ (a 6-LUT, often fracturable into two 5-input functions). Wide functions are built from trees of LUTs, and—because each LUT hop also costs routing delay—logic depth in LUTs is what your $F_{\max}$ ultimately trades against (§V).

Every LUT comes with a flip-flop. The basic logic cell pairs a LUT with a flip-flop (plus a carry bit); several cells form a *slice* and slices form a *CLB* (Xilinx) or *LAB* (Intel). The consequence a designer must internalize: because the flip-flop is bundled with the LUT, *registers are nearly free*, which is why FPGA designs pipeline aggressively to hit timing (§XIII).

TABLE II
Which FPGA resource to reach for.

Resource	Use it for
Flip-flop	pipeline registers, state; nearly free—pipeline freely
LUT	arbitrary combinational logic (any function of $\leq k$ inputs)
Carry chain	adders/subtractors, comparators
Distributed/LUT RAM	tiny, multi-ported memories (register files); async read
SRL	short fixed/variable-length shift registers, delay lines
Block RAM	buffers, FIFOs, caches, tables (sync read, $\geq$ 2-cycle)
DSP slice	multiply, multiply-accumulate, wide add

Dedicated hard blocks. Three hardened resources exist because building them from LUTs would be wasteful:

- **Carry chains** — dedicated fast carry logic so an N-bit add is one LUT per bit plus the fast chain, not a slow ripple through general logic.
- **Block RAM (BRAM)** — dedicated dual-port memories (tens of kilobits each) with *synchronous* read; enabling the optional output register gives a two-cycle read latency but higher $F_{\max}$. BRAMs have byte write-enables and can be a true or simple dual-port. Used for buffers, FIFOs, caches, coefficient tables.
- **DSP slices** — hardened multiply-accumulate units (a wide multiplier, an adder/accumulator, and pipeline registers). Any multiply, MAC, or wide add should map here rather than into LUTs.

Small and distributed memory. For a handful of words you do not spend a BRAM: a LUT can be reconfigured as *distributed* (LUT) RAM with *asynchronous* read—good for small, heavily-ported structures like a register file—or as a *shift-register LUT* (SRL) that packs a short shift register into a single LUT. Table II summarizes when to use which.

Fabric-scale structure. Clocks travel on dedicated global networks and clock regions (§VII); the largest devices are multi-die (stacked-silicon interconnect, SSI), partitioned into *super-logic regions* (SLRs), and a path crossing an SLR boundary pays a large fixed delay and must be registered on both sides. Above all, remember that on an FPGA *routing* delay, not gate delay, tends to dominate—placement matters as much as logic depth.

B. Questions

Q1. *What is inside a LUT, and how does it implement arbitrary logic?*

A. A k-input LUT is a $2^k \times 1$-bit memory (loaded from the configuration bitstream) read out by a 2^k:1 multiplexer whose select lines are the k inputs. Programming the memory with a function's truth table makes the LUT compute that function; since any truth table can be loaded, one LUT realizes any Boolean function of up to k inputs. Typical modern LUTs have $k = 6$.

Q2. *When would you store data in flip-flops, in distributed (LUT) RAM, and in block RAM?*

A. Flip-flops for small amounts of state that need to be read every cycle and possibly heavily ported (a few registers, pipeline stages). Distributed/LUT RAM for small memories—tens of words—especially when you need asynchronous read or multiple read ports cheaply (a small register file). Block RAM for anything from about a kilobit up (buffers, FIFOs, caches, lookup tables), accepting its synchronous, one- or two-cycle read latency in exchange for density. Choosing wrongly wastes resources: a BRAM for four words is profligate; a thousand words in flip-flops or LUT RAM will not fit.

Q3. *What operations should map to a DSP slice, and why care?*

A. Multiplies, multiply-accumulates, and wide additions. Care because a DSP slice is a hardened, fast, low-power unit with built-in pipeline registers; building a multiplier out of LUTs is large and slow. You steer inference with coding style and, if needed, attributes (`use_dsp`); getting a multiplier into a DSP rather than fabric is often the difference between meeting and missing timing.

Q4. *Why is a flip-flop considered "essentially free" on an FPGA, and what design habit follows?*

A. Because every logic cell already contains a flip-flop bundled with its LUT, so registering a signal usually consumes a resource you were given anyway. The habit that follows is aggressive pipelining: since added registers cost little area, you break long combinational paths into more, shorter stages to raise $F_{\max}$, accepting extra latency (§XIII).

Q5. *What is the carry chain and why does it matter?*

A. It is dedicated hardware that propagates carries between adjacent logic cells much faster than general routing could. It makes adders, subtractors, and comparators fast and compact—one LUT per bit plus the hard chain—so you should not, for instance, hand-build a ripple adder from generic gates; a `+` on a vector infers the carry chain automatically.

Q6. *Name the main differences between an FPGA and an ASIC from a designer's standpoint.*

A. An FPGA is a fixed menu of discrete, prefabricated resources (LUTs, FFs, BRAMs, DSPs) wired by programmable routing, so you compose from what the die provides and can run out of a specific resource while others sit idle; routing delay dominates and clock frequencies are a fraction of an ASIC's; but a build is minutes, and there is no fabrication cost or risk. An ASIC uses custom cells with gate-dominated delay, reaches far higher frequencies and densities, and can build exactly the structures it needs—at the cost of enormous, one-shot mask and fabrication expense. The practical upshot: FPGAs favor register-rich, pipelined, resource-aware designs and fast iteration.

Q7. *What is an SLR, and why does crossing one matter?*

A. A super-logic region is one die of a multi-die (stacked-silicon-interconnect) FPGA. A signal crossing the boundary between SLRs traverses a special interposer connection with a large, fixed delay, so such a crossing must be registered on both sides and kept off critical paths; accidental SLR crossings are a common cause of timing-closure trouble on large devices.

For a much deeper treatment of the fabric and how CPU-scale designs map onto it, see the RTL Processor Design series [14] and the vendor methodology guide [10]. Architecture knowledge pays off directly in the timing and pipelining sections that follow.

IX. Verilog and SystemVerilog Essentials

If timing and CDC are where interviewers test whether you *understand* hardware, HDL semantics are where they test whether you can *write* it without shooting yourself in the foot. The questions here are almost entirely about the gap between what the language lets you type and what the synthesizer actually builds. An interviewer asking these is checking that you know your code describes hardware, not a program that runs top to bottom.

A. Review

The single most important rule in Verilog is the discipline around **blocking** (`=`) versus **nonblocking** (`<=`) assignments. Cummings' rule, memorized by every practising designer, is: use nonblocking assignments in clocked (sequential) `always` blocks, use blocking assignments in combinational blocks, and never mix the two in one block [5]. The reason is in the simulation semantics. A nonblocking assignment samples its right-hand side immediately but defers the update of its left-hand side until the end of the current time step, *after* every other nonblocking RHS in the same step has been sampled. That deferred, parallel update is exactly how a bank of real flip-flops behaves—every register sees the *old* value of its neighbours on a clock edge, then all update together—so nonblocking assignment models registers correctly. Blocking assignment updates its LHS immediately, in program order, which models the combinational evaluation of a datapath but produces order-dependent, race-prone results if used for registers.

SystemVerilog gives the intent a name. Rather than the classic `always @(posedge clk)` and `always @(*)`, SystemVerilog provides `always_ff`, `always_comb`, and `always_latch`. These are not cosmetic: they tell the tools your intent, and the tools will *error* if the block's contents contradict it—an `always_comb` that infers a latch, or an `always_ff` without a clock edge, is flagged at elaboration rather than discovered as a silicon bug. Use them always.

Inferred latches are the most common accidental hardware. In a combinational block, if some output is not assigned on every path through the logic—a missing `else`, or a `case` without a `default` that does not cover every value—the semantics require the output to *hold* its previous value, which is a latch. The cure is to assign every output on every path: give every `if` an `else`, give every `case` a `default`, or assign default values to all outputs at the top of the block before the conditional logic overrides them.

A few smaller points round out the review. `wire` is a net driven by continuous assignment or a port; `reg` is a variable assigned in a procedural block (despite the name, a `reg` is not necessarily a register); SystemVerilog's `logic` replaces both in most code, being a 4-state variable usable in either context (it simply cannot be multiply driven). `parameter` and `localparam` make modules configurable; `generate` with `genvar` builds structural replication. And the recurring theme of every pitfall below is the same: your HDL is a *description of hardware*, and the synthesizer builds whatever your description implies, whether or not you meant it.

B. Questions

Q1. *What is the difference between blocking and nonblocking assignment, and what is the rule for using them?*

A. A blocking assignment (`=`) updates its left-hand side immediately, in program order within the block; a nonblocking

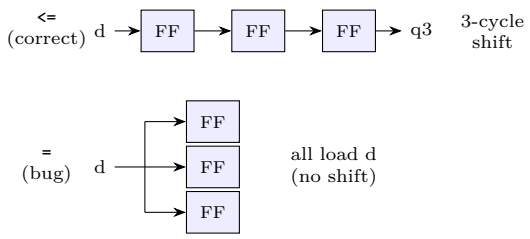

Fig. 7. Why the assignment operator matters inside a clocked block. Nonblocking (`<=`) builds the intended three-deep shift register; blocking (`=`) makes every stage load `d` in the same cycle, degenerating into three parallel one-cycle registers.

assignment (`<=`) samples its right-hand side immediately but defers the LHS update to the end of the time step, so all nonblocking updates in that step happen in parallel using each other's old values. The rule [5]: nonblocking in clocked sequential blocks (it correctly models the parallel update of flip-flops on a clock edge), blocking in combinational blocks (it models straight-line datapath evaluation), and never both in one block. Following the rule makes simulation match synthesis; breaking it produces race conditions where the result depends on the simulator's evaluation order.

Q2. *Why does using blocking assignment in a clocked block cause a bug? Give a concrete example.*

A. Consider a two-stage shift register. With nonblocking assignment, `q1 <= d; q2 <= q1;` both sample the old values, so `q2` gets the *previous* `q1`—two real flip-flops. With blocking assignment, `q1 = d; q2 = q1;` executes in order: `q1` becomes `d` immediately, then `q2 = q1` copies the *new* `q1`, so `q2` also becomes `d`—the second flip-flop collapses away and you have one register instead of two. Worse, whether it collapses can depend on the textual order of the statements, making the bug order-dependent. This is exactly why the rule exists.

Q3. *What is an inferred latch, how does it happen, and how do you prevent it?*

A. An inferred latch is a level-sensitive storage element the synthesizer creates because your combinational code does not specify an output's value on every execution path—so, to preserve the language's semantics, it must *hold* the old value, which requires memory. It happens from a missing `else`, or a `case` that neither covers all inputs nor has a `default`. Prevent it by fully specifying every output: default-assign all outputs at the top of the block, always pair `if` with `else`, and always give `case` a `default`. Using `always_comb` makes the tools flag the latch as an error instead of silently building it.

Q4. *What do `always_ff`, `always_comb`, and `always_latch` add over plain `always`?*

A. They declare intent and are checked against it. `always_comb` builds a combinational block with an automatically complete sensitivity list and errors if the code would infer a latch or a register; `always_ff` declares a clocked register and errors if there is no clock edge; `always_latch` declares an intentional latch (rare). The value is that a mismatch between what you meant and what you wrote becomes a compile-time error, not a subtle synthesis surprise.

Q5. *What is the difference between `reg`, `wire`, and `logic`?*

A. `wire` is a net: it must be driven continuously (by `assign` or a port) and can have multiple drivers resolved together. `reg` is a variable assigned procedurally inside an `always`/`initial` block; the name is historical and misleading—a `reg` synthesizes to a register only if it is assigned on a clock edge, otherwise to combinational logic or a latch. `logic` (SystemVerilog) is a 4-state type usable in place of both in single-driver contexts, which is nearly everywhere; modern code uses `logic` throughout and reserves `wire` for the rare multiply-driven net.

Q6. *(Spot the bug.) What hardware does Listing 11 produce, and why is that wrong? Fix it.*

A. It infers a latch on `y`. When `sel` is neither `2'b00` nor `2'b01`, no branch assigns `y`, so the semantics require it to hold—a latch. Two fixes work: add a `default`, or default-assign `y` before the `case` (Listing 12); using `always_comb` additionally turns the omission into a compile error.

```
always_comb begin
  case (sel)              // no default; y not always assigned
    2'b00: y = a;
    2'b01: y = b;         // sel = 2'b10 / 2'b11 -> y holds -> LATCH
  endcase
end
```

Listing 11. Broken: incomplete case infers a latch on `y` (SYSTEMVERILOG).

```
always_comb begin
  y = a;                  // default: y is assigned on every path
  case (sel)
    2'b01:   y = b;
    2'b10:   y = c;
    default: y = a;
  endcase
end
```

Listing 12. Fixed: default assignment covers every path (SYSTEMVERILOG).

Q7. *(Spot the bug.) Listing 13 is meant to be a 3-stage delay line but only delays by one cycle. Why, and how do you fix it?*

A. The clocked block uses blocking assignment, so within one evaluation `q1 = d` then `q2 = q1` then `q3 = q2` all propagate `d` straight through: every stage gets `d` in the same cycle and the two extra flip-flops vanish. Using nonblocking assignment (Listing 14) makes each stage sample the old value of the previous one, giving three real registers and a 3-cycle delay.

```
always_ff @(posedge clk) begin
  q1 = d;                 // blocking: q1 updates now,
  q2 = q1;                // q2 sees the NEW q1 = d,
  q3 = q2;                // q3 sees the NEW q2 = d  -> 1-cycle delay
end
```

Listing 13. Broken: blocking assignment collapses the shift register (SYSTEMVERILOG).

```
always_ff @(posedge clk) begin
  q1 <= d;                // all RHS sampled first, then all
  q2 <= q1;               // LHS update in parallel -> each stage
  q3 <= q2;               // sees the OLD previous stage -> 3-cycle
end
```

Listing 14. Fixed: nonblocking assignment gives three registers (SYSTEMVERILOG).

These pitfalls are the same class of question you will meet in §X for VHDL—the semantics differ but the lesson (know what your code builds) is identical.

X. VHDL Essentials

VHDL asks the same class of interview question as Verilog (§IX) —does the candidate know what the code builds?—but through a different and, for many candidates, subtler set of semantics. The signature VHDL question is the signal-versus-variable distinction, and it trips people precisely because VHDL's strong typing and verbosity lull you into treating it like software.

A. Review

The central VHDL concept is the difference between a **signal** and a **variable**. A *variable*, declared inside a process, updates

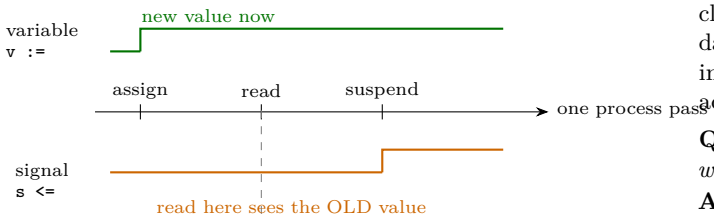

Fig. 8. Signal versus variable update within one process pass. A variable (`:=`) updates immediately, so a later read sees the new value; a signal (`<=`) updates only after the process suspends, so a read in the same pass sees the old value—the classic VHDL trap.

immediately when assigned (`:=`), just like a program variable; a subsequent read in the same process sees the new value. A *signal*, assigned with `<=`, does *not* update immediately: the new value is scheduled and takes effect only after the process suspends, one delta cycle later. The practical consequence, and the classic interview trap, is that if you assign a signal and then read it later in the *same* process execution, you read the *old* value, not the one you just assigned. Signals are how processes communicate and how registers are modelled (the deferred update mirrors a flip-flop's clock-edge behaviour, exactly as nonblocking assignment does in Verilog); variables are for local sequential computation within a process where you genuinely want immediate, step-by-step update.

The clocked-process idiom is the VHDL equivalent of `always_ff`: a process sensitive to the clock, with the body guarded by `if rising_edge(clk)`. Signal assignments inside it infer flip-flops. For synchronous designs this, plus a combinational process or concurrent assignments for the next-state and output logic, is the whole toolkit.

Use `numeric_std`, not `std_logic_arith`. Arithmetic on `std_logic_vector` should go through the IEEE `numeric_std` package with the `signed` and `unsigned` types, which are the standardized, portable, unambiguous way to do arithmetic. The old Synopsys `std_logic_arith`/`std_logic_unsigned` packages are non-standard, overlap ambiguously, and should be avoided in new code. Remember that `std_logic` is a nine-value type—`'0'`, `'1'`, high-impedance `'Z'`, unknown `'X'`, uninitialized `'U'`, and don't-care `'-'` among them—and that an `'X'` or `'U'` propagating through your simulation is a real bug signal, not noise to ignore.

Inferred latches and sensitivity-list bugs exist in VHDL too. A combinational process (one not clocked) that assigns a signal on only some paths of an `if` without an `else` infers a latch, exactly as in Verilog. And a combinational process whose sensitivity list omits a signal the process reads will simulate incorrectly (it will not re-evaluate when that signal changes) while synthesizing as though the list were complete—a classic sim/synth mismatch. VHDL-2008's `process(all)` builds the sensitivity list automatically and eliminates the bug; use it.

B. Questions

Q1. *What is the difference between a signal and a variable, and when do you use each?*

A. A variable (assigned with `:=`) updates immediately and is visible to later statements in the same process execution; a signal (assigned with `<=`) updates only after the process suspends, one delta cycle later, so a read later in the same execution sees the old value. Use signals to communicate between processes and to model registers—their deferred update matches a flip-flop clocking in new data while the rest of the logic still sees the old data. Use variables for local, immediate, sequential computation inside a single process, such as an accumulator you build up across a loop before driving the result onto a signal.

Q2. *In a process, you write `s <= a; ... if s = '1' then ...` where `s` was `'0'` before. Which branch is taken this pass?*

A. The `'0'` branch. The signal assignment `s <= a` only schedules `s`'s new value; it does not take effect until the process suspends, so the later `if s = '1'` still sees the *old* `s = '0'`. If you needed the updated value immediately within the process, you would use a variable (`v := a;` then test `v`). This is the single most common VHDL interview trap; naming it confidently signals real experience.

Q3. *Why should you use `numeric_std` rather than `std_logic_arith` / `std_logic_unsigned`?*

A. `numeric_std` is the IEEE-standardized arithmetic package; it defines `signed` and `unsigned` as distinct types over `std_logic`, so the signedness of every operation is explicit and portable across tools. `std_logic_arith` and its siblings are vendor-origin, non-standard, and define overlapping, ambiguous overloads (especially mixing signed and unsigned interpretations of the same vector), which leads to tool-dependent behaviour. New code should convert to `signed`/`unsigned` via `numeric_std` and back to `std_logic_vector` only at ports.

Q4. *What infers a latch in VHDL, and how do you avoid it?*

A. The same cause as in Verilog: a combinational (unclocked) process that does not assign a signal on every path—an `if` without a matching `else`, or a `case` without `when others`—forces the signal to hold its previous value, which is a latch. Avoid it by assigning every output on every path: give a default assignment at the top of the process, complete every `if` with `else`, and give every `case` a `when others`. (In a properly clocked process, holding a value is intended and correct —it is a register— so this concern is specifically about combinational processes.)

Q5. *What causes a simulation/synthesis mismatch from a sensitivity list, and what is the modern fix?*

A. In pre-2008 VHDL, a combinational process's sensitivity list is written by hand, and if it omits a signal the process reads, the simulator will not re-execute the process when that signal changes—so simulation shows stale outputs—whereas synthesis ignores the sensitivity list and builds the fully combinational function. The two disagree. The fix is VHDL-2008's `process(all)`, which makes the tool derive the complete sensitivity list automatically, guaranteeing the simulation matches the synthesized combinational logic.

Q6. *(Spot the bug.) The process in Listing 15 is meant to be combinational `y = a and b`. Why might simulation disagree with hardware, and how do you fix it?*

A. The sensitivity list contains only `a`, so when `b` changes (and `a` does not) the process does not re-evaluate and `y` keeps a stale value in simulation—yet synthesis builds a real AND gate that responds to both inputs. Simulation and hardware disagree. The fix is to list both inputs, or better, use `process(all)` (Listing 16).

```
comb : process (a)              -- missing b!
begin
  y <= a and b;                 -- sim won't re-run when b changes
end process;                    -- synth builds a real 2-input AND
```

Listing 15. Broken: incomplete sensitivity list causes a sim/synth mismatch (VHDL).

```
comb : process (all)            -- all read signals included
begin
```

```
  y <= a and b;                -- simulation now matches hardware
end process;
```

Listing 16. Fixed: VHDL-2008 process(all) derives the full list (VHDL).

Q7. *(Spot the bug.) Listing 17 tries to sum three values into* `sum` *but produces the wrong result. Why, and how do you fix it?*

A. `sum` is a signal, so each `sum <= sum + x` reads the *old* `sum` (the value before this process pass), not the running total from the lines above—only the last assignment survives, so `sum` becomes `c`, not `a+b+c`. Use a *variable* for the running accumulation, which updates immediately, and drive the signal once at the end (Listing 18).

```
process (all) begin
  sum <= sum + a;              -- each reads the OLD sum;
  sum <= sum + b;              -- earlier assignments are
  sum <= sum + c;              -- overwritten -> sum = c
end process;
```

Listing 17. Broken: signal reads see the old value, so only the last assignment wins (VHDL).

```
process (all)
  variable acc : unsigned(15 downto 0);
begin
  acc := (others => '0');
  acc := acc + a; acc := acc + b; acc := acc + c;
  sum <= acc;                  -- one signal assignment at the end
end process;
```

Listing 18. Fixed: a variable accumulates immediately; drive the signal once (VHDL).

The signal/variable distinction and the sensitivity-list trap are the two VHDL answers an interviewer is most likely to probe; rehearse them until they are reflexive, as with the timing fundamentals of §V.

XI. RTL Coding Idioms and Exercises

Whiteboard rounds almost always ask you to write one of a small set of standard blocks from memory. None is hard, but under pressure candidates fumble the details—an off-by-one in a counter, an edge detector that emits a two-cycle pulse, a debouncer with no synchronizer. This section drills the highest-yield idioms as short exercises. For each, read the prompt, write it yourself, then compare with the model answer. All solutions assume a single clock `clk`, an active-high synchronous reset, and rising-edge logic; asynchronous inputs are assumed already synchronized (§VI).

A. Review

The blocks worth having in muscle memory are the **counter** (with enable and load), the **shift register**, the **edge detector**, the **debouncer**, the **gray-code counter**, the **priority encoder**, and the **round-robin arbiter**. Each is a few lines, but each hides one classic mistake, called out below.

B. Counter with Enable and Load

Q1. *Write an N-bit up-counter with a synchronous load and a count enable.*

A. The only subtlety is precedence: reset over load over enable. Test that in order in the clocked process.

```
module counter #(parameter int N = 8) (
  input  logic         clk, rst, load, en,
  input  logic [N-1:0] d,
  output logic [N-1:0] q
);
  always_ff @(posedge clk)
    if      (rst)  q <= '0;
    else if (load) q <= d;
    else if (en)   q <= q + 1'b1;
endmodule
```

Listing 19. Counter with load and enable (SYSTEMVERILOG).

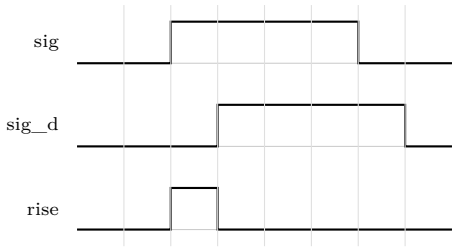

Fig. 9. Rising-edge detection. `rise` is high for exactly one cycle where `sig` is high and its one-cycle-delayed copy is still low.

C. Rising-Edge Detector

Q2. *Write a rising-edge detector that emits a single one-cycle pulse when its input goes high.*

A. Register the input one cycle, then AND the current value with the inverse of the delayed value: `rise = sig & ~sig_d`. The classic mistake is forgetting the register and comparing the signal with itself, or producing a level instead of a one-cycle pulse. If `sig` comes from another clock domain, synchronize it first (§VI)—edge-detecting a raw asynchronous signal will occasionally double- or zero-count on metastability. The waveform is shown in Fig. 9.

```
library ieee; use ieee.std_logic_1164.all;
entity edge_det is
  port (clk, rst, sig : in std_logic; rise : out std_logic);
end entity;
architecture rtl of edge_det is
  signal sig_d : std_logic := '0';
begin
  process (clk) begin
    if rising_edge(clk) then
      if rst = '1' then sig_d <= '0'; else sig_d <= sig; end if;
    end if;
  end process;
  rise <= sig and not sig_d;   -- one-cycle pulse
end architecture;
```

Listing 20. Rising-edge detector (VHDL).

```
module edge_det (input logic clk, rst, sig, output logic rise);
  logic sig_d;
  always_ff @(posedge clk) sig_d <= rst ? 1'b0 : sig;
  assign rise = sig & ~sig_d;   // one-cycle pulse
endmodule
```

Listing 21. Rising-edge detector (SYSTEMVERILOG).

D. Switch Debouncer

Q3. *Write a debouncer that ignores a mechanical input until it has been stable for* `STABLE` *clock cycles.*

A. Count consecutive cycles for which the (synchronized) input differs from the current debounced output; when the count reaches the threshold, adopt the new level and clear the count. Any glitch resets the count. The mistake to avoid is debouncing the *raw* pin without first passing it through a synchronizer (§VI).

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity debounce is
  generic (CW : natural := 16);          -- STABLE = 2**CW cycles
  port (clk, rst, noisy : in std_logic; clean : out std_logic);
end entity;
architecture rtl of debounce is
  signal cnt : unsigned(CW-1 downto 0) := (others => '0');
  signal q   : std_logic := '0';
begin
  process (clk) begin
    if rising_edge(clk) then
      if rst = '1' then cnt <= (others=>'0'); q <= '0';
      elsif noisy /= q then
        cnt <= cnt + 1;
        if cnt = to_unsigned(2**CW-1, CW) then q <= noisy; cnt <= (others
      =>'0'); end if;
      else cnt <= (others => '0'); end if;
    end if;
```

```
  end process;
  clean <= q;
end architecture;
```

Listing 22. Counter-based debouncer (VHDL).

```
module debounce #(parameter int CW = 16) (
  input logic clk, rst, noisy, output logic clean);
  logic [CW-1:0] cnt; logic q;
  always_ff @(posedge clk)
    if (rst)            begin cnt <= '0; q <= 1'b0; end
    else if (noisy != q) begin
      cnt <= cnt + 1'b1;
      if (&cnt) begin q <= noisy; cnt <= '0; end   // stable long enough
    end else cnt <= '0;                            // glitch: restart
  assign clean = q;
endmodule
```

Listing 23. Counter-based debouncer (SYSTEMVERILOG).

E. Gray-Code Counter

Q4. *Write a counter whose output is gray-coded (exactly one bit changes per step).*

A. The clean way is a binary counter plus the identity gray $=$ bin $\oplus$ (bin $\gg 1$); keeping the binary count internally also makes the value easy to use elsewhere. This is exactly the pointer a dual-clock FIFO crosses safely (§VI, §XII).

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity gray_cnt is
  generic (N : natural := 4);
  port (clk, rst, en : in std_logic;
        gray : out std_logic_vector(N-1 downto 0));
end entity;
architecture rtl of gray_cnt is
  signal bin : unsigned(N-1 downto 0) := (others => '0');
begin
  process (clk) begin
    if rising_edge(clk) then
      if rst = '1' then bin <= (others => '0');
      elsif en = '1' then bin <= bin + 1; end if;
    end if;
  end process;
  gray <= std_logic_vector(bin xor ('0' & bin(N-1 downto 1)));
end architecture;
```

Listing 24. Gray-code counter (VHDL).

```
module gray_cnt #(parameter int N = 4) (
  input logic clk, rst, en, output logic [N-1:0] gray);
  logic [N-1:0] bin;
  always_ff @(posedge clk)
    if (rst)     bin <= '0;
    else if (en) bin <= bin + 1'b1;
  assign gray = bin ^ (bin >> 1);
endmodule
```

Listing 25. Gray-code counter (SYSTEMVERILOG).

F. Priority Encoder

Q5. *Write a priority encoder that returns the index of the lowest set bit of an N-bit request vector, and a valid flag.*

A. Scan from the high index down to zero so the last (lowest) set bit wins, or use the bit-isolation trick `req & (~req + 1)` to peel off the lowest set bit directly. A combinational `for` loop is the readable answer:

```
module prienc #(parameter int N = 8) (
  input  logic [N-1:0]         req,
  output logic [$clog2(N)-1:0] idx,
  output logic                 valid
);
  always_comb begin
    idx = '0; valid = 1'b0;
    for (int i = N-1; i >= 0; i--)
      if (req[i]) begin idx = i[$clog2(N)-1:0]; valid = 1'b1; end
  end
endmodule
```

Listing 26. Priority encoder / find-first-one (SYSTEMVERILOG).

G. Round-Robin Arbiter

Q6. *Write a round-robin arbiter: given an N-bit request vector, grant exactly one request (one-hot), rotating priority so no requester starves.*

A. Keep a one-hot *base* marking the current highest-priority slot. Among requests at or above the base, grant the lowest; if there are none, wrap and grant the lowest overall. After each grant, advance the base past the winner. The lowest-set-bit primitive is $x \,\&\, (\bar{x} + 1)$. This is the idiom that most often trips candidates—the wrap-around and the base update are where bugs hide.

```
module rr_arbiter #(parameter int N = 4) (
  input  logic         clk, rst,
  input  logic [N-1:0] req,
  output logic [N-1:0] grant
);
  logic [N-1:0] base;                      // one-hot highest-priority slot
  logic [N-1:0] hi_req, hi_g, lo_g;
  assign hi_req = req & ~(base - 1'b1);    // requests at/above base
  assign hi_g   = hi_req & (~hi_req + 1'b1); // lowest set of those
  assign lo_g   = req    & (~req    + 1'b1); // lowest set overall (wrap)
  assign grant  = (|hi_req) ? hi_g : lo_g;
  always_ff @(posedge clk)
    if (rst)        base <= {{(N-1){1'b0}}, 1'b1};
    else if (|grant) base <= {grant[N-2:0], grant[N-1]};  // next slot
endmodule
```

Listing 27. Round-robin arbiter (SYSTEMVERILOG).

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity rr_arbiter is
  generic (N : natural := 4);
  port (clk, rst : in std_logic;
        req   : in  std_logic_vector(N-1 downto 0);
        grant : out std_logic_vector(N-1 downto 0));
end entity;
architecture rtl of rr_arbiter is
  signal base, g : unsigned(N-1 downto 0);
  signal hi, hi_g, lo_g : unsigned(N-1 downto 0);
  signal r : unsigned(N-1 downto 0);
begin
  r    <= unsigned(req);
  hi   <= r and not (base - 1);
  hi_g <= hi and (not hi + 1);              -- lowest set of masked
  lo_g <= r  and (not r  + 1);              -- lowest set overall
  g    <= hi_g when hi /= 0 else lo_g;
  grant <= std_logic_vector(g);
  process (clk) begin
    if rising_edge(clk) then
      if rst = '1' then base <= to_unsigned(1, N);
      elsif g /= 0 then base <= g(N-2 downto 0) & g(N-1); end if;
    end if;
  end process;
end architecture;
```

Listing 28. Round-robin arbiter (VHDL).

These idioms recur inside larger blocks throughout the rest of the guide—the arbiter in bus interfaces (§XIV), the gray counter in FIFOs (§XII), the edge detector everywhere. Being able to produce them without hesitation is a large fraction of a coding round.

XII. Memory and FIFO Design

Memories and FIFOs are the most common "design a block" request after the FSM. The FIFO in particular is a favourite because it touches everything: memory inference, pointer arithmetic, full/empty generation, and—for the dual-clock case—clock-domain crossing (§VI). This section reviews on-chip memory, then walks the synchronous FIFO and the classic depth-sizing problem.

A. Review

FPGA storage comes in three tiers (§VIII): flip-flops for a handful of bits, **distributed (LUT) RAM** for small memories with asynchronous read, and **block RAM (BRAM)** for kilobit-and-larger arrays. BRAM has properties an interviewer expects you to know: it is *dual-port*, its read is *synchronous* (the address

is registered, so data appears the cycle after the address, and an optional output register adds a second cycle for timing—so plan for one- or two-cycle read latency), it supports byte write enables, and its behaviour when the same address is read and written in the same cycle (read-first / write-first / no-change) is a configured mode you must choose deliberately. You get a BRAM by writing the standard template—a 2-D array with a *registered* read—and *not* doing anything that forces asynchronous read or too many ports; the `ram_style` attribute can force or forbid the choice.

A **FIFO** is a memory plus a write pointer, a read pointer, and full/empty logic. In a **synchronous FIFO** both pointers live in one clock domain and full/empty come from a straight comparison. In an **asynchronous FIFO** the two pointers live in different clock domains and must be gray-coded and synchronized before comparison (§VI); this is treated in depth in the companion series [14] and by Cummings [4].

B. Questions

Q1. *How do you write RTL that infers a block RAM, and what commonly prevents the inference?*

A. Declare a 2-D array (`reg [W-1:0] mem [DEPTH]`) and read it *through a register*—the read address or the read data must be clocked, so the tool maps it to BRAM's synchronous-read port. Inference is commonly blocked by: an *asynchronous* (combinational) read, which forces distributed RAM or flip-flops; needing more read/write ports than a BRAM offers (at most two); initializing or resetting the entire array (BRAM contents cannot be reset by logic, only initialized at configuration); or reading and writing the same array in incompatible ways. When in doubt, apply `ram_style = "block"` and check the synthesis report to confirm the tool obeyed.

Q2. *What is the read latency of a block RAM, and why?*

A. One cycle with just the input (address) register, or two cycles with the optional output register enabled. The address is captured on a clock edge and the data emerges on the next edge; the extra output register is a pipeline stage that shortens the BRAM-to-fabric path and is usually needed to hit timing at speed (§V). A candidate who assumes combinational (same-cycle) BRAM read has a latent bug: the pipeline downstream will be off by a cycle.

Q3. *When would you use a synchronous FIFO versus an asynchronous FIFO?*

A. Use a **synchronous** FIFO when the writer and reader share a clock—for rate matching, buffering bursts, or decoupling a producer and consumer in one domain; it is simple and cheap. Use an **asynchronous** FIFO only when the two sides are in *different, unrelated clock domains*: it is the standard, safe way to move a data stream across a clock boundary (§VI), but it costs the gray-coded pointer synchronizers and their latency, so do not reach for it when one clock would do.

Q4. *How are the full and empty flags generated in a synchronous FIFO?*

A. From the write and read pointers. Two common schemes: keep an explicit occupancy *count* (increment on write, decrement on read, and compare against 0 and `DEPTH`); or use pointers one bit *wider* than the address and compare—equal low bits with equal MSBs means empty, equal low bits with differing MSBs means full (the "extra-MSB" trick, which also makes the async version's gray comparison work). The count scheme is the clearest to write under pressure and is used below.

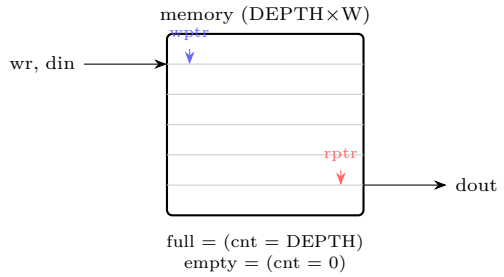

Fig. 10. A synchronous FIFO: one memory, a write pointer, a read pointer, and an occupancy count driving `full`/`empty`.

Q5. *A writer produces a burst of 256 words back-to-back, one word per cycle at 200 MHz. The reader can only accept one word every two cycles (effectively 100 MHz) during the burst. How deep must the FIFO be to avoid overflow?*

A. During the burst the FIFO fills at (write rate − read rate). Over the time it takes to write 256 words, the reader removes $256 \times \frac{100}{200} = 128$ words, so the FIFO accumulates $256 - 128 = 128$ words. The minimum depth is therefore **128**, and in practice you round up to a power of two (128 is already one) and add margin for pointer-synchronizer latency if it is an async FIFO. The general rule: depth $\geq N_{\text{burst}} \left(1 - \frac{R_{\text{read}}}{R_{\text{write}}}\right)$, plus crossing latency.

C. Exercise: A Synchronous FIFO

Q6. *Write a parameterized synchronous FIFO with `full` and `empty` flags.*

A. Use a count for the flags and first-word-fall-through (combinational) read so data is available before the read strobe. Guard writes with `!full` and reads with `!empty` so the pointers never run past each other. Fig. 10 shows the structure.

```
module sync_fifo #(parameter int W = 8, DEPTH = 16,
                   localparam int AW = $clog2(DEPTH)) (
  input  logic         clk, rst,
  input  logic         wr,  input  logic [W-1:0] din,  output logic full,
  input  logic         rd,  output logic [W-1:0] dout, output logic empty
);
  logic [W-1:0] mem [DEPTH];
  logic [AW-1:0] wptr, rptr;
  logic [AW:0]   cnt;                       // 0 .. DEPTH
  assign full  = (cnt == DEPTH);
  assign empty = (cnt == 0);
  assign dout  = mem[rptr];                 // first-word-fall-through
  always_ff @(posedge clk)
    if (rst) begin wptr <= '0; rptr <= '0; cnt <= '0; end
    else begin
      if (wr && !full)  begin mem[wptr] <= din; wptr <= wptr + 1'b1; end
      if (rd && !empty) rptr <= rptr + 1'b1;
      cnt <= cnt + (wr && !full) - (rd && !empty);
    end
endmodule
```

Listing 29. Parameterized synchronous FIFO (SYSTEMVERILOG).

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity sync_fifo is
  generic (W : natural := 8; DEPTH : natural := 16);
  port (clk, rst : in std_logic;
        wr : in std_logic; din : in std_logic_vector(W-1 downto 0); full :
      out std_logic;
        rd : in std_logic; dout : out std_logic_vector(W-1 downto 0); empty :
       out std_logic);
end entity;
architecture rtl of sync_fifo is
  constant AW : natural := integer(ceil(log2(real(DEPTH))));
  type mem_t is array (0 to DEPTH-1) of std_logic_vector(W-1 downto 0);
  signal mem : mem_t;
  signal wptr, rptr : unsigned(AW-1 downto 0) := (others => '0');
  signal cnt : unsigned(AW downto 0) := (others => '0');
  signal do_wr, do_rd : std_logic;
begin
  full  <= '1' when cnt = DEPTH else '0';
  empty <= '1' when cnt = 0     else '0';
  do_wr <= wr and not (full  when cnt = DEPTH else '0');  -- see note
```

```
  do_rd <= rd and not (empty when cnt = 0     else '0');
  dout  <= mem(to_integer(rptr));                          -- FWFT
  process (clk) begin
    if rising_edge(clk) then
      if rst = '1' then wptr<=(others=>'0'); rptr<=(others=>'0'); cnt<=(
      others=>'0');
      else
        if wr = '1' and cnt /= DEPTH then mem(to_integer(wptr))<=din; wptr<=
      wptr+1; end if;
        if rd = '1' and cnt /= 0     then rptr<=rptr+1; end if;
        if    (wr='1' and cnt/=DEPTH) and not (rd='1' and cnt/=0) then cnt<=
      cnt+1;
        elsif (rd='1' and cnt/=0)     and not (wr='1' and cnt/=DEPTH) then
      cnt<=cnt-1;
        end if;
      end if;
    end if;
  end process;
end architecture;
```

Listing 30. Parameterized synchronous FIFO (VHDL).

For the dual-clock version, the pointers become gray-coded and each is synchronized into the opposite domain before the full/empty comparison; that design, and why gray coding is what makes it safe, is the subject of the companion series' clock-domain-crossing volume [14] and is a frequent senior-level follow-up to this exercise.

XIII. Pipelining, Latency, and Throughput

Pipelining is the lever an FPGA designer reaches for most often, because on an FPGA flip-flops are nearly free (§VIII) and the interconnect, not the gates, dominates delay. An interviewer probes this topic to see whether you understand the trade it makes—more clock cycles of latency in exchange for a higher clock frequency—and whether you can carry a *valid* qualifier alongside the data so the pipeline stays correct.

A. Review

A combinational function has some propagation delay t_{logic} from its inputs to its outputs. If that delay exceeds the clock period the design cannot meet timing (§V). **Pipelining** inserts register stages that cut the long combinational path into shorter segments, so each segment fits in one clock period and the achievable $F_{\max}$ rises (Fig. 11). The three terms an interviewer will insist you keep straight are:

- **Latency** — the number of clock cycles (or time) from an input entering the pipeline to its corresponding result emerging. Pipelining *increases* latency: an N-stage pipeline has a latency of N cycles.
- **Throughput** — the rate at which results emerge once the pipeline is full, usually one result per clock. Pipelining is what lets a deep computation still produce a result every cycle.
- **Initiation interval (II)** — the number of cycles between successive inputs the pipeline can accept. A fully pipelined datapath has II = 1 (a new input every cycle); a resource that must be reused has II > 1.

The distinction that trips candidates up is that **latency and throughput are independent**. A 10-stage floating-point unit may have a latency of 10 cycles yet a throughput of one operation per cycle—ten operations are in flight at once, each in a different stage. Confusing "it takes 10 cycles" with "it does one every 10 cycles" is a classic error.

Retiming is the automatic version of this: the synthesis tool moves existing registers across combinational logic to balance the delay of each stage, without changing the design's cycle-accurate behaviour. You enable it by providing registers for the tool to move (for example, clustering several output registers that retiming can then push back into the logic) and by not blocking it with `dont_touch`. Retiming rebalances; it does not create registers you did not write.

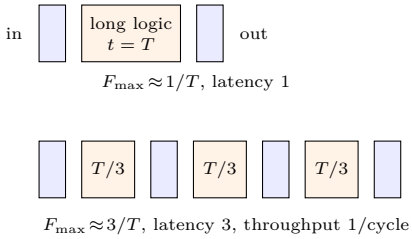

Fig. 11. Pipelining cuts one long combinational path into shorter stages. The clock can run roughly three times faster, at the cost of two extra cycles of latency; throughput stays one result per cycle.

Finally, **not everything can be pipelined**. A computation with a *feedback* dependency—an accumulator whose next value depends on its own previous value, or any recurrence—cannot be pipelined along the feedback path, because the result must be available before the next input can be processed. The loop's latency sets a hard lower bound on the II. Recognizing a recurrence is often the real content of a pipelining interview question.

B. Questions

Q1. *Define latency and throughput, and explain how they differ.*
A. Latency is the delay from an input entering a block to its result emerging, measured in clock cycles; throughput is the rate at which results emerge once the block is running, usually results per cycle. They are independent: a deeply pipelined unit can have high latency (many cycles to produce any one result) and still have full throughput (a new result every cycle) because many operations are in flight simultaneously, one per stage. The mistake to avoid is treating "takes N cycles" as "one result every N cycles"—that is only true for an unpipelined, II-N resource.

Q2. *How does pipelining raise the maximum clock frequency, and what does it cost?*
A. It cuts a long combinational path into shorter segments separated by registers, so the longest register-to-register delay—which sets $F_{\max}$ (§V)—shrinks. If you split a path of delay T into three balanced stages, each stage is about $T/3$, so $F_{\max}$ roughly triples. The costs are: additional latency (one cycle per added stage), a few more flip-flops (cheap on an FPGA, §VIII), and the design complexity of keeping control and data aligned—any control signal or side path must be delayed by the same number of stages, which is why you propagate a valid bit through matching registers.

Q3. *What is retiming, and how do you let the tool do it?*
A. Retiming is a synthesis transformation that relocates existing registers across combinational logic to equalize the delay of each pipeline stage, preserving the design's cycle-accurate behaviour. You enable it by giving the tool registers to move—for instance, writing several registers on the output of a wide multiplier so retiming can push them back into the partial-product logic—and by not pinning those registers with `dont_touch` or `keep`. Retiming only moves registers you wrote; it will not invent the pipeline depth for you.

Q4. *You have pipelined a datapath. How do you know which stages contain valid results, for example after a flush or during start-up?*
A. Propagate a one-bit *valid* qualifier through a shift register that has exactly the same depth as the data pipeline, driven

by the input-valid signal and cleared on reset or flush. The output is meaningful only when its corresponding valid bit is set. This "valid pipeline" is the standard idiom; the exercise below shows it. For back-pressured pipelines you additionally need the valid/ready handshake of §XIV and a stall that freezes all stages together (via a common clock enable, §VII).

Q5. *Give an example of a computation you cannot pipeline, and explain why.*

A. An accumulator, `sum <= sum + x`, or any feedback recurrence where the next output depends on the previous output. The new value cannot be computed until the previous one is available, so inserting registers in the feedback loop would change the arithmetic, not just delay it. The loop's combinational delay divided by the number of registers already in the loop sets the minimum clock period; if that is too slow you must restructure the algorithm (for example, *C*-slowing by interleaving independent streams, or a tree/carry-save reassociation) rather than simply adding pipeline stages.

C. Exercise: A Pipelined Multiply-Add with Valid Propagation

A common whiteboard task is "pipeline $p = a \times b + c$ so it meets timing, and tell me when the output is valid." The multiply and the add each get their own stage, with an input-register stage in front, giving three cycles of latency. A valid bit rides through three registers alongside the data. Listings 31 and 32 show the paired solutions.

```vhdl
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity pipe_mac is
  generic (W : natural := 16);
  port (clk, rst_n, in_valid : in  std_logic;
        a, b, c   : in  signed(W-1 downto 0);
        p         : out signed(2*W-1 downto 0);
        out_valid : out std_logic);
end entity;
architecture rtl of pipe_mac is
  signal a1, b1, c1 : signed(W-1 downto 0);       -- stage 1 regs
  signal mul2       : signed(2*W-1 downto 0);      -- stage 2 product
  signal c2         : signed(2*W-1 downto 0);
  signal vld        : std_logic_vector(2 downto 0); -- valid pipe
begin
  process (clk) begin
    if rising_edge(clk) then
      if rst_n = '0' then
        vld <= (others => '0');
      else
        -- stage 1: register inputs
        a1 <= a; b1 <= b; c1 <= c;
        -- stage 2: multiply, carry c along
        mul2 <= a1 * b1;
        c2   <= resize(c1, 2*W);
        -- stage 3: add
        p    <= mul2 + c2;
        -- valid shifts through the same 3 stages
        vld  <= vld(1 downto 0) & in_valid;
      end if;
    end if;
  end process;
  out_valid <= vld(2);
end architecture;
```

Listing 31. Three-stage pipelined multiply-add with valid propagation (VHDL).

```systemverilog
module pipe_mac #(parameter int W = 16) (
  input  logic                clk, rst_n, in_valid,
  input  logic signed [W-1:0] a, b, c,
  output logic signed [2*W-1:0] p,
  output logic                out_valid
);
  logic signed [W-1:0]   a1, b1, c1;      // stage 1
  logic signed [2*W-1:0] mul2, c2;        // stage 2
  logic [2:0]            vld;             // valid pipeline
  always_ff @(posedge clk) begin
    if (!rst_n) vld <= '0;
    else begin
      a1 <= a; b1 <= b; c1 <= c;          // stage 1: register inputs
      mul2 <= a1 * b1;                    // stage 2: multiply
      c2   <= $signed({{W{c1[W-1]}}, c1}); // sign-extend c
      p    <= mul2 + c2;                  // stage 3: add
      vld  <= {vld[1:0], in_valid};       // shift valid
    end
  end
  assign out_valid = vld[2];
endmodule
```

Listing 32. The same pipelined multiply-add (SYSTEMVERILOG).

The teaching points to state aloud: the multiply gets its own stage because it is the deepest logic (it maps to a DSP slice whose internal pipeline registers you are effectively lighting up, §VIII); `c` is delayed through a matching register so it aligns with the product at the adder; and the valid pipeline is exactly three deep so `out_valid` tracks the data. Deeper treatment of pipelining for frequency appears in the companion RTL series [14].

XIV. Interfaces and Handshake Protocols

Blocks must talk to one another, and the interface between them is where a surprising number of bugs live. Interviewers favour this topic because it is where correctness (never dropping or duplicating a beat) meets timing (not letting a combinational `ready` form a long path). The *valid/ready* handshake and its buffering are the ideas to master.

A. Review

The dominant on-chip streaming interface is the **valid/ready handshake** (the basis of AXI-Stream). A producer drives `valid` and the payload; a consumer drives `ready`; a single beat of data transfers on a rising clock edge exactly when `valid` and `ready` are both high (Fig. 12). Two rules make it composable, and an interviewer will expect you to know them:

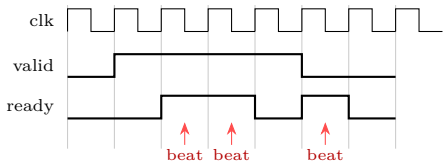

Fig. 12. The valid/ready handshake. A beat transfers only on edges where both signals are high (arrows). While `valid` is high but `ready` is low, the producer stalls and holds its payload.

- **A producer must not wait for `ready` to assert `valid`.** `valid` may be raised whenever data is available; making it depend on `ready` creates a combinational loop between the two sides.
- **Once `valid` is asserted, the payload must remain stable until the beat completes.** The producer may not change or withdraw the data while waiting for `ready`; the consumer relies on it holding.

Back-pressure is what `ready` provides: when a consumer cannot accept data it deasserts `ready`, the producer stalls, and if that producer is itself the consumer of an upstream stage, the stall ripples backwards through the pipeline. This is how a slow sink safely throttles a fast source without losing data.

The subtlety is timing. If `ready` is a purely combinational function of the downstream state, and it is ANDed with `valid` to enable the producer, a long chain of connected blocks can build a combinational path that runs the whole length of the pipeline—destroying $F_{\max}$ (§V). The fix is the **skid buffer** (also called a register slice): a two-entry buffer that registers both the payload and the handshake so neither `valid` nor `ready` passes through in one combinational hop, while still sustaining one beat per cycle. The second entry is the "skid" slot that catches the beat already in flight when the consumer suddenly deasserts `ready`. The exercise below builds one.

TABLE III
THE THREE COMMON SERIAL INTERFACES AT A GLANCE.

	SPI	**I²C**	**UART**
Wires	4 (SCLK,MOSI,MISO,CS)	2 (SCL,SDA)	2 (TX,RX)
Clock	shared (master)	shared	none (async)
Address	chip select	on-bus addr	none (point-point)
Speed	high (10s of MHz)	low–mod	low

AXI is the bus you are most likely to be asked about. Full AXI has five *independent* channels—write address, write data, write response, read address, read data—each a valid/ready stream, which is why crossing AXI between clock domains is done channel by channel with small FIFOs (§VI). **AXI-Lite** is a stripped-down single-beat variant for control registers; **AXI-Stream** is just the payload valid/ready handshake with no addressing, for data flows. You are not usually asked to implement AXI on a whiteboard, but you should be able to name the channels and explain why their independence matters.

The classic **serial interfaces** round out the topic (Table III): SPI (full-duplex, a clock plus separate data lines and a chip select, fast, no addressing on the wire), I²C (two wires shared by many devices, open-drain, address-based, slower), and UART (asynchronous—no shared clock—start/stop framed bytes at an agreed baud rate).

B. Questions

Q1. *Describe the valid/ready handshake and its two rules.*
A. The producer asserts `valid` with a payload; the consumer asserts `ready` when it can accept; a beat transfers on a clock edge where both are high. Rule one: the producer must not condition `valid` on `ready` (that would create a combinational loop and violates the protocol). Rule two: once `valid` is high, the payload must stay stable until the transfer completes. Together these let arbitrary producers and consumers be connected without dropping or duplicating data, and give back-pressure for free.

Q2. *What is back-pressure, and how does it propagate?*
A. Back-pressure is a consumer signalling "not now" by de-asserting `ready`, which stalls the producer. If that producer is a pipeline stage, its own upstream `ready` deasserts in turn, so the stall ripples backward, one stage per cycle in a registered pipeline. It lets a slow sink throttle a fast source safely, at the cost of the buffering needed to absorb data already in flight while the stall propagates.

Q3. *Why would you insert a skid buffer, and what does it do?*
A. To break timing. If `ready` and `valid` are combinational across many connected blocks, they form one long path that limits F_{max}. A skid buffer registers both the handshake and the payload so neither crosses the boundary combinationally, while still passing one beat per cycle at full throughput. It needs two storage slots: one for the beat being presented and one "skid" slot to catch the beat that was already in flight when the downstream `ready` dropped, so no data is lost.

Q4. *What are the channels of full AXI, and why are they independent?*
A. Write address (AW), write data (W), write response (B), read address (AR), and read data (R). Each is its own valid/ready stream. Independence lets reads and writes proceed concurrently, lets addresses be issued ahead of data (pipelining transactions), and—practically for us—lets a clock-domain crossing be done one channel at a time with a small asynchronous FIFO per channel (§VI), since the protocol never requires cross-channel timing coordination.

Q5. *Compare SPI, I²C, and UART.*
A. SPI is synchronous and full-duplex: a master-driven clock plus separate transmit and receive lines and a chip-select per slave; fast, simple, but uses more wires and has no on-wire addressing. I²C uses just two open-drain wires (clock and data) shared by many devices, with addresses sent on the bus; it is slower and needs pull-ups and arbitration, but scales to many devices on two pins. UART is asynchronous—no shared clock—sending framed bytes (start bit, data, optional parity, stop bit) at a pre-agreed baud rate between two endpoints; simplest wiring, but both sides must agree on the rate and it is point-to-point.

Q6. *How do you carry a valid/ready stream across a clock-domain boundary?*
A. Do not try to synchronize `valid` and `ready` independently with two-flop synchronizers—they would lose their mutual timing and drop or duplicate beats. Use an asynchronous FIFO (§VI, §XII): the producer writes into it in its clock domain, the consumer reads in the other, and the FIFO's gray-coded pointers make the crossing safe while the full/empty flags provide the back-pressure on each side.

C. Exercise: A Two-Entry Skid Buffer

Listings 33 and 34 implement the register slice: it accepts an upstream beat whenever it has room, and presents a registered beat downstream, so both handshake directions are registered and full throughput is preserved.

```vhdl
library ieee; use ieee.std_logic_1164.all;
entity skid is
  generic (W : natural := 32);
  port (clk, rst_n : in std_logic;
        s_valid : in  std_logic; s_ready : out std_logic;
        s_data  : in  std_logic_vector(W-1 downto 0);
        m_valid : out std_logic; m_ready : in  std_logic;
        m_data  : out std_logic_vector(W-1 downto 0));
end entity;
architecture rtl of skid is
  signal buf   : std_logic_vector(W-1 downto 0);
  signal out_q : std_logic_vector(W-1 downto 0);
  signal has_buf, out_vld : std_logic := '0';
begin
  s_ready <= (not out_vld) or m_ready;  -- room if output free or draining
  m_valid <= out_vld;  m_data <= out_q;
  process (clk) begin
    if rising_edge(clk) then
      if rst_n = '0' then out_vld <= '0'; has_buf <= '0';
      else
        if m_ready = '1' or out_vld = '0' then       -- output slot advances
          if has_buf = '1' then
            out_q <= buf; out_vld <= '1'; has_buf <= '0';
          else
            out_q <= s_data; out_vld <= s_valid;
          end if;
        elsif s_valid = '1' and has_buf = '0' then    -- catch the skid beat
          buf <= s_data; has_buf <= '1';
        end if;
      end if;
    end if;
  end process;
end architecture;
```

Listing 33. Two-entry skid buffer / register slice (VHDL).

```systemverilog
module skid #(parameter int W = 32) (
  input  logic         clk, rst_n,
  input  logic         s_valid, output logic s_ready,
  input  logic [W-1:0] s_data,
  output logic         m_valid, input  logic m_ready,
  output logic [W-1:0] m_data
);
  logic [W-1:0] buf_q, out_q;
  logic         has_buf, out_vld;
  assign s_ready = ~out_vld | m_ready;
  assign m_valid = out_vld;
  assign m_data  = out_q;
  always_ff @(posedge clk) begin
    if (!rst_n) begin out_vld <= 1'b0; has_buf <= 1'b0; end
    else begin
```

```
      if (m_ready || !out_vld) begin
        if (has_buf) begin out_q <= buf_q; out_vld <= 1'b1; has_buf <= 1'b0;
      end
        else         begin out_q <= s_data; out_vld <= s_valid;
      end
      end else if (s_valid && !has_buf) begin
        buf_q <= s_data; has_buf <= 1'b1;   // skid slot
      end
    end
  end
endmodule
```

Listing 34. The same skid buffer (SYSTEMVERILOG).

Say aloud why the skid slot is needed: in the cycle the downstream drops `ready`, the upstream may still be presenting a beat that the buffer has already accepted (because it advertised `ready`); that beat lands in `buf` and is replayed when the output drains. Without it you would either lose that beat or have to make `s_ready` combinationally depend on `m_ready`, reintroducing the long path. The companion RTL series [14] and the UltraFast methodology guide [10] treat register slices in more depth.

XV. Verification and Testbenches

In industry, more engineer-hours go into verifying hardware than designing it, so an interviewer—especially for a mid or senior role—will check that you treat verification as a first-class activity rather than an afterthought. The themes are: make the testbench *self-checking*, drive it with a mix of directed and random stimulus, encode invariants as *assertions*, and use *coverage* to know when you are done.

A. Review

A testbench instantiates the design under test (DUT), generates a clock and reset, applies stimulus, and—this is the part beginners skip—*checks the results automatically.* A **self-checking** testbench compares the DUT's output against an independent reference (a golden model, a reference function, or simply the known-correct answer for directed vectors) and reports pass or fail, so a regression of hundreds of tests needs no human to read waveforms. Eyeballing a waveform proves nothing repeatable and does not scale.

Stimulus comes in two styles. **Directed** tests hand-craft specific scenarios—reset behaviour, the empty and full corners of a FIFO, a back-to-back burst—and are essential for the cases you know are tricky. **Constrained-random** tests generate legal but unpredictable input within declared constraints, and are how you find the bugs you did *not* think of; they are only useful in combination with a self-checking reference and with coverage to tell you what the randomness actually exercised.

Assertions encode design invariants as executable checks. SystemVerilog assertions (SVA) come in two forms: *immediate* assertions, evaluated like a procedural statement at one instant, and *concurrent* assertions, evaluated against a clock over time and able to express temporal properties ("if `valid` is high and `ready` is low, then next cycle `valid` is still high and the data is unchanged"). Assertions catch a violation *at its source*, the cycle it happens, instead of as a mismatched output many cycles later, which is why they dramatically shorten debug. Good properties to assert include handshake stability (§XIV), that a one-hot signal really is one-hot, that a FIFO never overflows or underflows, and that no control signal goes unknown (`X`) after reset.

Coverage answers "are we done?" *Code coverage* measures which lines, branches, and expressions the tests exercised; *functional coverage* measures which specified scenarios (via cover points and cross-coverage) occurred. The point an interviewer wants to hear: **100% code coverage does not mean the design is correct**—it only means every line ran, not that every interesting combination of conditions was tried or that the outputs were right. Functional coverage plus self-checking is what closes the gap.

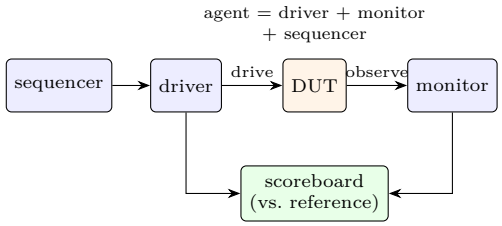

Fig. 13. A constrained-random verification environment. The driver and monitor form an agent around the DUT interface; the scoreboard checks observed output against a reference. This is the shape UVM standardizes.

UVM, the Universal Verification Methodology, is the industry framework for building reusable constrained-random environments. You should be able to name its pieces at a high level (Fig. 13): an *agent* wraps a *driver* (drives the interface from transactions), a *monitor* (observes the interface into transactions), and a *sequencer* (feeds transactions to the driver); a *scoreboard* compares observed results against a reference; and a *factory* lets components be overridden without editing them. For an FPGA role you are rarely asked to write UVM on the spot, but you should know why it exists—reuse and constrained-random at scale.

B. Questions

Q1. *What distinguishes a good testbench from a bad one?*
A. A good testbench is *self-checking* and *reproducible*: it compares the DUT against an independent reference and prints an unambiguous pass/fail, so it can run unattended in a regression and a failure points at what broke. A bad testbench requires a human to inspect waveforms, which does not scale, is not repeatable, and silently "passes" when nobody is looking. Secondary marks of quality: it randomizes within legal constraints to find unexpected bugs, it measures coverage so you know what was exercised, and it tests the corner cases (reset, empty/full, back-to-back, back-pressure) explicitly.

Q2. *When would you use directed versus constrained-random stimulus?*
A. Directed tests for the scenarios you already know are risky or are required by the spec—reset, specific corner cases, a particular protocol sequence—because they are deterministic and pinpoint. Constrained-random for breadth: it explores combinations you did not enumerate and finds the bugs you did not anticipate, but only pays off with a self-checking reference to judge correctness and functional coverage to confirm what it actually hit. Real verification uses both: directed for the known, random for the unknown.

Q3. *What is an assertion? Give a concrete example.*
A. An assertion is an executable statement of a design invariant that fires the moment the invariant is violated, at the source rather than downstream. For a valid/ready interface, a concurrent SVA property states that stalled data must hold stable: if `valid` is high and `ready` is low, then on the next cycle `valid` is still high and the payload is unchanged (Listing 36). Because

it checks every cycle against the clock, a violation is reported exactly when and where it happens, which is far faster to debug than a mismatched output many cycles later.

Q4. *What is coverage, and why does 100% code coverage not mean you are done?*

A. Coverage measures how much of the design/space the tests exercised. Code coverage (line, branch, expression, toggle) tells you which RTL executed; functional coverage (cover points and crosses) tells you which specified scenarios occurred. 100% code coverage only means every line *ran*—not that the outputs were *correct* (that is the checker's job), and not that every meaningful *combination* of conditions was tried (that is functional coverage's job). A design can have full code coverage and still be wrong on an untested interaction, so you close verification with self-checking plus functional coverage, not code coverage alone.

Q5. *What does UVM provide that a hand-written testbench does not?*

A. A standard, reusable structure for constrained-random verification: transaction-level stimulus (sequences and sequencers), reusable interface agents (driver plus monitor), a scoreboard pattern for self-checking, phasing to coordinate build/run/check across many components, and a factory that lets you override or configure components without editing them. The payoff is reuse across projects and scaling to large random environments; the cost is significant boilerplate, which is why for a small FPGA block a focused self-checking testbench with assertions is often the pragmatic choice.

Q6. *How would you verify a FIFO?*

A. Model the FIFO's expected behaviour with a reference queue in the testbench: on every accepted write, push the data; on every accepted read, pop and compare against the DUT's output—that is the self-checking core. Drive it with constrained-random writes and reads (including independent random backpressure on both sides) to exercise arbitrary fill levels, and add directed tests for the corners: read from empty must not underflow, write to full must not overflow, simultaneous read and write at both empty and full, and reset mid-stream. Assert the invariants (never overflow, never underflow, full and empty never both true) and cover that the FIFO actually reached empty, full, and the almost-flags.

C. Exercise: A Self-Checking Snippet and an Assertion

Listing 35 is the checking core of a self-checking testbench in SystemVerilog—a reference model and a compare—and Listing 36 is the handshake-stability property referred to above.

```
// Reference for a DUT that should output prev_in delayed by one cycle.
logic [7:0] ref_q;
always_ff @(posedge clk)
  if (!rst_n) ref_q <= '0;
  else        ref_q <= dut_in;         // golden behaviour

// Self-check: every cycle after reset, DUT output must match the reference.
int errors = 0;
always_ff @(posedge clk)
  if (rst_n && (dut_out !== ref_q)) begin
    $error("mismatch @%0t: dut=%02x ref=%02x", $time, dut_out, ref_q);
    errors++;
  end

final $display("TEST %s (%0d errors)", (errors==0)?"PASSED":"FAILED", errors)
      ;
```

Listing 35. The self-checking core of a testbench: reference model plus compare (SYSTEMVERILOG).

```
// If a beat is offered (valid) but not accepted (!ready),
// then next cycle valid is still high and the payload is unchanged.
property p_stable;
  @(posedge clk) disable iff (!rst_n)
    (valid && !ready) |=> (valid && $stable(data));
endproperty

assert property (p_stable)
  else $error("handshake payload changed while stalled");
```

Listing 36. A concurrent SVA property: stalled handshake data must stay stable (SYSTEMVERILOG).

The points to make: the reference model is deliberately independent of the DUT's implementation (here a one-cycle delay), the compare uses `!==` so an `X` on the output is caught, and the `final` block turns the run into an unambiguous pass/fail. The SVA property encodes rule two of the valid/ready handshake (§XIV) so any violation is flagged the instant it occurs. The SystemVerilog standard [12] and the companion series [14] cover assertions and UVM in depth.

XVI. Debug, Bring-Up, and Timing Closure

This section is where an interviewer probes whether you have actually put a design on hardware, because the questions have no textbook answer—they test judgement built from experience. The two recurring scenarios are "it works in simulation but fails on the board" and "it fails to meet timing," and having a crisp, ordered approach to each is worth more than any single fact.

A. Review

On-chip debug. Because you cannot probe internal FPGA nodes with an oscilloscope, vendors provide embedded instruments: an *integrated logic analyzer* (ILA / ChipScope) captures selected internal signals into block RAM on a trigger condition and streams them out over JTAG; a *virtual I/O* (VIO) lets you drive and observe slow control signals from the host. You mark signals for capture with an attribute such as `mark_debug`. The costs are real and worth stating: the ILA consumes BRAM proportional to capture width times depth, adds logic and routing that can perturb the very timing you are chasing (the observer effect), and its capture depth is limited, so you trigger narrowly.

"Works in simulation, fails on hardware." This phrase should immediately cue a short list of suspects, because functional simulation cannot reproduce them:

- **Clock-domain crossing** (§VI). Zero-delay simulation never produces metastability, so a missing or wrong synchronizer passes in sim and fails intermittently on hardware—the classic case, and the first thing to suspect for an *intermittent* failure.
- **Reset** (§VII). An asynchronous reset released near a clock edge, or logic that depends on a reset value the hardware does not actually guarantee, behaves in sim (where reset is ideal) but not on the board.
- **Uninitialized state / X-optimism.** Simulation may propagate `X` pessimistically or optimistically differently from real silicon, which powers up to definite (if unknown) values; a design that relied on a register's initial value can diverge.
- **Timing.** If the design did not actually meet timing (or a false path was mis-constrained), the hardware samples stale data where simulation, which ignores delays unless told otherwise, does not.

The method is to narrow by symptom: an *intermittent* or temperature/voltage-sensitive failure points at CDC or metastability; a failure *only right after configuration or reset* points at the reset scheme; a failure that *tracks clock frequency* (works when you slow the clock) points at a setup-timing problem; and a hard, repeatable functional failure points at a plain logic bug or a mis-constraint. Then instrument the suspected area with an ILA and capture around the trigger.

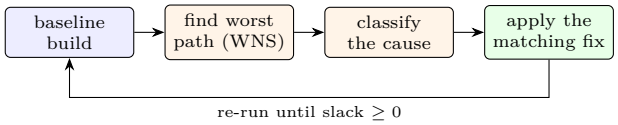

Fig. 14. Timing closure is a triage loop: baseline, find the worst negative-slack path, classify its cause, apply the matching fix, and re-run—not a single transformation applied blindly.

Approaching a timing failure. Timing closure is a triage loop, not a single trick (Fig. 14). Read the timing report, find the path with the worst negative slack (WNS), and *classify* it, because the fix follows from the cause:

- **Too many logic levels** → pipeline it (§XIII) or restructure the logic into a balanced tree.
- **High fan-out net** → replicate the driver / register-duplicate so each copy drives fewer loads and shorter routes.
- **Long routing / placement spread** → floorplan the offending logic into a pblock, or let physical optimization move it.
- **A false or multicycle path reported as failing** → it is a *constraint* bug; add the correct exception (§VI, §V) rather than changing logic.

Fix the highest-impact category, re-run, and repeat. Chasing every path at once, or over-constraining, wastes time; work the worst path and its siblings first.

Bring-up order. When a fresh board arrives, bring it up in dependency order so each step rests on a verified foundation: confirm the clocks are present and the PLL/MMCM locks; confirm reset asserts and releases cleanly; confirm the simplest sign of life (a heartbeat LED or a UART banner, §XIV); then exercise the data path incrementally. Jumping straight to the full application when you have not confirmed the clock is locked wastes hours.

B. Questions

Q1. *Your design passes every simulation but fails on the board. What do you suspect, and how do you narrow it down?*

A. The failure is almost certainly something simulation cannot model: a clock-domain crossing (no synchronizer, or a multi-bit bus crossed unsafely), a reset problem (async release, or reliance on an initial value the hardware does not guarantee), an uninitialized-state / X-optimism divergence, or a real timing violation. I narrow by symptom: intermittent or temperature-sensitive → CDC/metastability; failure right after reset/configuration → reset scheme; failure that disappears when I slow the clock → setup timing; hard and repeatable → a logic bug or mis-constraint. Then I put an ILA on the suspected signals and capture around a trigger to see what actually happens.

Q2. *How do you approach a path that fails timing?*

A. Read the report and take the worst-negative-slack path first, then classify it. Too many logic levels between registers means pipeline or restructure into a balanced tree; a high-fan-out net means replicate the driver; long routing means floorplan or run physical optimization; and a false/multicycle path reported as failing is a constraint bug, fixed by adding the right timing exception, not by touching logic. I fix the biggest contributor, re-run, and iterate. I avoid over-constraining and avoid randomly adding pipeline stages before I know the cause.

Q3. *What is an ILA, and what does using one cost you?*

A. An integrated logic analyzer is an embedded instrument that captures chosen internal signals into block RAM on a trigger and reads them out over JTAG, letting you see internal behaviour you cannot probe physically. The costs: it consumes BRAM proportional to the number of signals times the capture depth, it adds logic and routing that can shift timing on the paths you are observing (the observer effect), and its depth is finite, so you must trigger narrowly on the event of interest rather than capturing everything.

Q4. *How would you close timing on a critical combinational path?*

A. First understand why it is critical. If it is deep logic, add pipeline registers to cut it (accepting the added latency, §XIII) or rebalance it into a shallower tree; enable retiming so the tool can rebalance the stages. If a single net has enormous fan-out, duplicate its driving register so each copy serves fewer loads over shorter routes. If the endpoints are placed far apart, constrain them into a pblock to shorten routing. And if the path should not have been timed at all—a quasi-static configuration register or a genuine multicycle path—the right fix is the correct timing exception, not more logic.

Q5. *You have a brand-new board. What is a sensible bring-up order?*

A. Bottom-up, so each step stands on a verified one. First confirm power and that the clocks are present and the PLL/MMCM locks—nothing works without a clock. Next confirm reset asserts and, importantly, releases cleanly. Then get the simplest possible sign of life: a blinking LED driven by a counter, or a fixed UART banner, which proves the clock, reset, and a trivial data path all work. Only then bring up the real data path incrementally, adding an ILA where you need visibility. Trying to run the full application before the clock is confirmed locked is the most common way to waste a debug day.

Debug and closure reward method over cleverness; the companion RTL series [14] and the UltraFast methodology guide [10] give the fuller closure playbook, and §XVII puts several of these ideas to work on concrete problems.

XVII. Classic Whiteboard Problems

The problems in this section recur across FPGA interviews with remarkable consistency; a prepared candidate has seen every one of them before. Work each by hand before reading the solution, and—because the interviewer is grading your reasoning—narrate the approach as you go: state the invariant, sketch the timing or the state diagram, then write the RTL. The solutions below emphasize the *insight* the problem tests, which is what you must reproduce under pressure, not memorized code.

A. Clocking and Counters

Q1. *Divide a clock by three with a 50% duty cycle.*

A. The insight the interviewer wants first: a single-edge counter can only place edges at rising clock edges, so its output is high for an *integer* number of clock periods—giving $\frac{1}{3}$ (33%) or $\frac{2}{3}$ (66%) duty, never 50%. A 50% duty on a period of $3T_{\text{clk}}$ requires the output to be high for $1.5\,T_{\text{clk}}$, and the only way to transition at a half-period point is to *use both clock edges*. The standard construction runs one divide-by-three waveform off the rising edge and an identical one off the falling edge—so they are offset

by half a period—and combines them. Listing 37 shows the count-based divide-by-three (the part you should always get right); the 50% version ORs it with a negative-edge copy.

```
module div3 (input logic clk, input logic rst_n, output logic clk3);
  logic [1:0] cp, cn;   logic p, n;
  // rising-edge mod-3 counter and a 1-of-3 high pulse
  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) begin cp <= 2'd0; p <= 1'b0; end
    else begin cp <= (cp==2'd2) ? 2'd0 : cp+2'd1;
               p  <= (cp != 2'd2); end          // high 2 of 3 rising edges
  // falling-edge copy, offset half a period
  always_ff @(negedge clk or negedge rst_n)
    if (!rst_n) begin cn <= 2'd0; n <= 1'b0; end
    else begin cn <= (cn==2'd2) ? 2'd0 : cn+2'd1;
               n  <= (cn != 2'd2); end
  assign clk3 = p & n;   // AND of two 66%-duty, half-period-shifted waves = 50%
endmodule
```

Listing 37. Divide-by-three; OR with a negedge copy for 50% duty (SYSTEMVERILOG).

```
process (clk, rst_n) begin
  if rst_n = '0' then cp <= 0; p <= '0';
  elsif rising_edge(clk) then
    if cp = 2 then cp <= 0; else cp <= cp + 1; end if;
    if cp /= 2 then p <= '1'; else p <= '0'; end if;   -- high 2 of 3
  end if;
end process;
-- a falling-edge process produces n identically; clk3 <= p and n;
```

Listing 38. Divide-by-three counter core (VHDL).

A crucial caveat to say out loud: a divided clock generated in the fabric like this should be treated as a clock only with care—on a real FPGA you would instead use a clock-enable at the base clock, or an MMCM, rather than route a LUT-generated clock (§VII).

Q2. *Generate a single-cycle pulse from an input that stays high for many cycles.*

A. This is edge detection: register the input and compare with its delayed copy. A rising-edge pulse is `sig & ~sig_q`, where `sig_q` is `sig` delayed one clock. If `sig` is *asynchronous* it must first be synchronized (§VI) before the edge detector, or you will occasionally emit a runt pulse. The full idiom, with both edges, is in §XI.

B. Encoding and Bit Manipulation

Q3. *Convert an n-bit binary value to Gray code and back.*

A. Binary to Gray is one line: each Gray bit is the XOR of the binary bit with the binary bit above it, i.e. `g = b ^ (b >> 1)`. The inverse is an XOR prefix-scan: the Gray MSB copies straight down, and each lower binary bit is the XOR of the Gray bit with the binary bit *just computed* above it, `b[i] = g[i] ^ b[i+1]`. Gray's defining property—exactly one bit changes between consecutive values—is why it is the safe encoding for the FIFO pointers that cross clock domains (§VI).

```
function automatic [N-1:0] bin2gray(input [N-1:0] b);
  return b ^ (b >> 1);
endfunction
function automatic [N-1:0] gray2bin(input [N-1:0] g);
  logic [N-1:0] b;
  b[N-1] = g[N-1];
  for (int i = N-2; i >= 0; i--) b[i] = g[i] ^ b[i+1];
  return b;
endfunction
```

Listing 39. Binary↔Gray conversion (SYSTEMVERILOG).

```
-- binary to gray
gray <= bin xor ('0' & bin(N-1 downto 1));
-- gray to binary (XOR prefix scan)
process (gray) begin
  b(N-1) := gray(N-1);
  for i in N-2 downto 0 loop b(i) := gray(i) xor b(i+1); end loop;
  bin <= b;
end process;
```

Listing 40. Binary↔Gray conversion (VHDL).

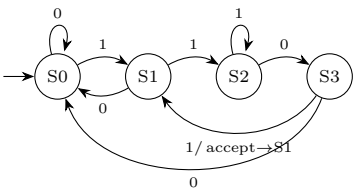

Fig. 15. Moore FSM for the overlapping detector `1101`. On accepting, the final `1` overlaps the next match, so control returns to `S1`, not `S0`.

Q4. *Find the position of the least-significant set bit of a word (design a priority encoder).*

A. A priority encoder scans from one end and reports the first asserted bit. The neat trick for the *lowest* set bit is that `x & (-x)` isolates it (two's complement, §III), after which a small case or a one-hot-to-binary encoder gives its index. In hardware you usually just write the priority `casez`/`if` chain; the full encoder appears in §XI. State the ambiguity aloud: define what the output is when the word is zero (a valid flag, or an all-ones index).

Q5. *Count the number of set bits in a word (population count).*

A. For a small word, a balanced adder tree: sum the bits pairwise, then those partial sums, and so on in $\lceil \log_2 n \rceil$ levels—each level is a layer of small adders, and the whole thing is a purely combinational reduction that you pipeline (§XIII) if it fails timing on a wide word. For large words the interviewer may want the classic divide-and-conquer bit-twiddling (mask-and-add by 1s, 2s, 4s), which is the same adder tree expressed with shifts and masks.

C. State Machines

Q6. *Design an FSM that detects the overlapping sequence 1101 on a serial input.*

A. A Moore detector needs one state per matched prefix: `S0` (nothing), `S1` (saw `1`), `S2` (saw `11`), `S3` (saw `110`); the output asserts on the transition into the accepting condition after `S3` sees the final `1`. The subtlety that *overlapping* introduces (Fig. 15) is where you go on a mismatch: after accepting `1101`, the trailing `1` is itself the start of a new match, so you return to `S1` (not `S0`); and from `S2` (`11`) a `1` keeps you in `S2`. Getting those two mismatch/overlap edges right is the whole point of the problem—enumerate the next state for *both* input values in *every* state (§IV).

D. Clock Domains

Q7. *Build a glitch-free clock multiplexer that selects between two asynchronous clocks.*

A. A plain combinational mux on two clocks is unsafe: if the select changes while either clock is high, the output can produce a runt or a glitch that downstream flip-flops miscount. The glitch-free structure switches only when the outgoing clock is safely low, and guarantees the two clocks are never simultaneously enabled. For each clock domain, synchronize an "enable me" request into that domain with two flip-flops, gate it so a clock is enabled only when the *other* clock's enable has been observed de-asserted, and AND each clock with its (negative-latched) enable. The cross-coupling—each side's enable depends on the other side being off—is what enforces break-before-make. Listing 41 sketches one side; in practice you use the device's dedicated `BUFGMUX`/`BUFGCTRL` primitive, which implements exactly this, and you should name it.

```systemverilog
// en0 asserted only if the other clock's enable is observed low;
// negedge latch ensures the AND-gate opens while clk0 is low.
logic sel_sync0_meta, sel_sync0;
always_ff @(posedge clk0) begin
  sel_sync0_meta <= sel_bit & ~en1_seen;    // want clk0 AND clk1 is off
  sel_sync0      <= sel_sync0_meta;
end
logic en0;
always_ff @(negedge clk0) en0 <= sel_sync0; // change enable while clk0 low
assign clk0_gated = clk0 & en0;             // real design: BUFGMUX/BUFGCTRL
// clk_out = clk0_gated | clk1_gated;  (the two are never both enabled)
```

Listing 41. Glitch-free clock-mux enable for one clock (mirror for the other) (SYSTEMVERILOG).

Q8. *Safely pass a 16-bit counter value from a slow clock domain to a fast one.*

A. Because it is a *counter* that changes by one each step, Gray-code it so only one bit changes per update, then run each Gray bit through a two-flop synchronizer in the destination domain and convert back to binary (§VI). If the value were *arbitrary* rather than a counter, Gray would not help—you would use a request/acknowledge handshake (hold the data stable, cross a single valid bit, capture on the far side) or an asynchronous FIFO for a stream. Saying *why* Gray works here but not in general (single bit changes vs. arbitrary multi-bit change) is the discriminating answer.

E. Sizing

Q9. *A producer writes bursts of B words at one word per write-clock; the consumer drains at one word every R write-clocks on average. How deep must the FIFO be?*

A. Model the worst case during a burst: over the burst duration the producer writes B words while the consumer removes only $\lfloor B/R \rfloor$, so the FIFO must hold the difference, roughly $B - \lfloor B/R \rfloor$ words, plus margin for the pointer-synchronization latency if it is an asynchronous FIFO (§VI). The method the interviewer is checking is that you compute *accumulated* write-minus-read over the burst window, round up, and add CDC slack—not that you recite a formula. Always ask for the read and write *rates* and the *burst length*; a FIFO depth question with no burst bound is under-specified, and pointing that out scores well (§XII).

XVIII. Rapid-Fire Question Bank

This section is for the night before. The questions are short, the answers are one or two sentences, and the goal is recall speed, not depth—each item points back to the section where it is developed properly. Read the question, answer it in your head, and check. If any answer here is not instant, revisit the corresponding section rather than memorizing the one-liner.

A. Timing

Q1. *What is setup time?*

A. The interval *before* a clock edge during which a flip-flop's data input must be stable (§V).

Q2. *What is hold time?*

A. The interval *after* a clock edge during which the data input must stay stable.

Q3. *Can you fix a hold violation by slowing the clock?*

A. No—hold is independent of clock period. You fix it by adding delay on the short path or correcting clock skew.

Q4. *Can you fix a setup violation by slowing the clock?*

A. Yes—a longer period gives the data more time to arrive; that is exactly what lowering $F_{\max}$ does.

Q5. *What is $F_{\max}$?*

A. The highest clock frequency at which all setup checks pass, i.e. the reciprocal of the longest register-to-register path delay.

Q6. *What are WNS and TNS?*

A. Worst negative slack (the single worst path) and total negative slack (the sum over all failing paths); both zero or positive means timing is met.

Q7. *What is clock skew?*

A. The difference in clock-edge arrival time between two flip-flops; it helps or hurts setup and hold in opposite directions.

Q8. *What is clock uncertainty?*

A. A timing margin the tool subtracts to account for jitter and skew when checking setup/hold.

Q9. *What is a multi-cycle path?*

A. A path the design allows more than one clock period to traverse; you tell the tool with a multicycle constraint so it is not falsely reported as failing.

B. Metastability and CDC

Q10. *What is metastability?*

A. A flip-flop that samples a signal violating setup/hold can enter an unstable state that takes an unbounded time to resolve (§VI).

Q11. *Why does a two-flop synchronizer help?*

A. The second flop grants the first's output nearly a full clock period to resolve, raising the mean time between failures exponentially.

Q12. *Can you two-flop-synchronize a multi-bit bus bit by bit?*

A. No—the bits resolve independently, so the captured word can be neither the old nor the new value. Use Gray code or a handshake.

Q13. *What does the `ASYNC_REG` attribute do?*

A. It tells the tools a flip-flop is a synchronizer stage, so they place the stages tightly and do not optimize or retime them.

Q14. *May combinational logic sit between synchronizer stages?*

A. No—nothing may read the first stage's output before the second stage samples it, or the settling budget is lost.

Q15. *How do you cross a stream of data between clock domains?*

A. An asynchronous FIFO with Gray-coded read/write pointers (§VI, §XII).

Q16. *How do you cross a single control pulse?*

A. A toggle synchronizer: convert the pulse to a level toggle, synchronize the level, and edge-detect on the far side.

Q17. *What constraint goes on a CDC path?*

A. `set_false_path` or, better, `set_max_delay -datapath_only`, so static timing does not try to time the asynchronous capture.

C. HDL Semantics

Q18. *Blocking or non-blocking for clocked logic?*

A. Non-blocking (`<=`)—it models parallel register update; blocking (`=`) is for combinational logic (§IX).

Q19. *What infers a latch?*

A. An incompletely specified combinational block—a missing `else` or a `case` without `default` leaves a signal unassigned on some path.

Q20. *How do you prevent an inferred latch?*

A. Assign every output on every path; in SystemVerilog use `always_comb`, in VHDL default the outputs at the top of the process.

Q21. *Signal vs. variable in VHDL?*

A. A variable updates immediately within the process; a signal takes its new value only after the process suspends (§X).

Q22. *`always_ff` vs. `always_comb`?*

A. `always_ff` declares clocked (registered) logic; `always_comb` declares combinational logic and flags accidental latches.

Q23. *Why `numeric_std` over `std_logic_arith`?*

A. `numeric_std` is the IEEE standard with unambiguous `signed`/`unsigned` types; the older Synopsys packages are non-standard and can clash.

Q24. *What causes a sim/synth mismatch from a sensitivity list?*

A. An incomplete sensitivity list on a combinational process—simulation misses events synthesis includes; use `always_comb` or `process(all)`.

Q25. *`reg` vs. `logic` in SystemVerilog?*

A. `logic` is a 4-state type usable for both procedural assignment and continuous drive; `reg` is its Verilog predecessor and does not imply a register.

D. FPGA Architecture

Q26. *What is a LUT?*

A. A k-input lookup table: a 2^k-bit truth-table memory read by a mux tree, implementing any function of k inputs (§VIII).

Q27. *What is block-RAM read latency?*

A. One cycle for the synchronous read, plus a second if the optional output register is enabled—so plan for two cycles.

Q28. *Registers, distributed RAM, or block RAM—how do you choose?*

A. A few bits: registers; small and multi-ported or needing asynchronous read: distributed (LUT) RAM; kilobits or more: block RAM.

Q29. *What maps to a DSP slice?*

A. Multiplies and multiply-accumulates; inferring a DSP saves fabric and runs faster than a LUT multiplier.

Q30. *Why is a flip-flop essentially free on an FPGA?*

A. Every logic cell already contains one, so pipelining costs area you have already paid for—pipeline freely (§XIII).

Q31. *What is the carry chain for?*

A. Dedicated fast carry logic that makes adders and counters fast without consuming general routing.

Q32. *ASIC vs. FPGA, one difference that matters most?*

A. On an FPGA the programmable routing dominates delay, so where the tool places endpoints matters as much as logic depth.

Q33. *What is an SLR?*

A. A super-logic region—one die of a multi-die FPGA; crossing between SLRs costs a large fixed delay and must be registered.

E. RTL Design

Q34. *One-hot or binary FSM encoding on an FPGA?*

A. Usually one-hot: it shortens decode logic and flip-flops are cheap (§IV).

Q35. *Moore or Mealy—which output is glitch-free?*

A. Moore: its output depends only on the registered state, so it can be registered clean; Mealy reacts a cycle earlier but exposes combinational input-to-output paths.

Q36. *What is a skid buffer?*

A. A two-entry buffer that registers a valid/ready handshake so back-pressure is not in the combinational path, sustaining one beat per cycle (§XIV).

Q37. *State the valid/ready handshake rule.*

A. Transfer occurs when both are high; a producer must not wait for ready before asserting valid, and must hold the payload until the beat completes.

Q38. *How do you save power without gating a clock?*

A. Use a clock enable on the flip-flops (and `BUFGCE` for coarse gating), never a fabric-gated clock (§VII).

Q39. *Recommended reset scheme?*

A. Asynchronous assertion, synchronous de-assertion, with a per-domain reset synchronizer.

Q40. *How do you raise $F_{\max}$ on a long combinational path?*

A. Pipeline it—insert register stages to cut logic depth—accepting added latency (§XIII).

Q41. *What is retiming?*

A. The tool moving registers across combinational logic to balance stage delays without changing function.

F. Verification and Debug

Q42. *What makes a good testbench?*

A. It is self-checking—it computes the expected result and flags mismatches automatically, rather than relying on waveform inspection (§XV).

Q43. *Directed vs. constrained-random testing?*

A. Directed targets specific known cases; constrained-random explores the space and, with coverage, finds cases you did not think of.

Q44. *Immediate vs. concurrent SVA assertion?*

A. Immediate checks a condition at a point in procedural code; concurrent checks a temporal property sampled on a clock.

Q45. *What is functional coverage?*

A. A measure of which interesting scenarios the tests actually exercised, independent of code coverage.

Q46. *Your design works in simulation but hangs in hardware—first suspects?*

A. A clock-domain-crossing bug, a reset problem, or a timing failure—none of which a zero-delay simulation exposes (§XVI).

Q47. *What is an ILA?*

A. An integrated logic analyzer: on-chip logic that captures internal signals into block RAM on a trigger for debug.

Q48. *How do you approach a failing timing path?*

A. Read the report, find the worst path, classify the cause—logic depth, high fanout, or routing—and apply the matching fix (pipeline, replicate, floorplan).

Q49. *What is the single most common CDC bug?*

A. Passing a multi-bit value through per-bit synchronizers instead of Gray-coding it or using a handshake.

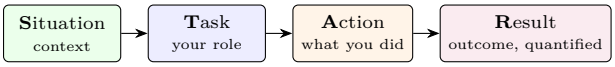

Fig. 16. The STAR structure. The Result, quantified, is the part candidates most often drop and interviewers most want to hear.

XIX. Behavioral and Experience Questions

Candidates who prepare exhaustively for timing and CDC often walk into the behavioral round cold, and it costs them. Hardware teams are small, schedules are long, and a tape-out or a bring-up is unforgiving of someone who cannot communicate, take feedback, or own a mistake—so this round is not a formality. The good news is that it rewards preparation just as much as the technical rounds do: the questions are predictable, and a handful of well-chosen stories, told well, cover almost all of them.

A. The STAR Structure

The single most useful tool is the STAR structure—Situation, Task, Action, Result (Fig. 16). Most behavioral answers fail in one of two ways: they are a vague generality ("I'm good at debugging") with no concrete instance, or they are a shapeless ramble with no point. STAR fixes both. Set the **Situation** in one sentence (the project, the stakes); state your specific **Task** (what *you*, not the team, were responsible for); describe the **Action** you took in enough technical detail to be credible but without drowning the listener; and end with a concrete **Result**, quantified if you can ("closed timing at 250 MHz," "cut the bug rate in the CDC paths to zero," "shipped two weeks early"). The result is the part candidates most often omit and interviewers most want.

Prepare a story bank. Before the interview, write down four to six concrete episodes from your work—a hard bug, a design tradeoff you drove, a disagreement you resolved, a mistake you made, a project you are proud of, a time you learned something fast. Each should be a real, specific event with a technical spine and a clear outcome. Most behavioral questions are then a matter of selecting the right story and telling it in STAR form, rather than inventing one under pressure.

B. Questions and How to Answer Them

The answers below are *guidance on structure*, not scripts—an interviewer can tell a memorized answer instantly. For each, note what is really being assessed.

Q1. *Tell me about the hardest bug you have debugged.*
A. Assessing: your debugging methodology and your honesty about difficulty. Pick a genuinely hard, ideally hardware-specific bug (a CDC glitch, a reset-release race, an intermittent timing failure), and walk the STAR arc with emphasis on *Action*: how you narrowed it down—what you hypothesized, what you measured (an ILA capture, a targeted assertion), how you ruled things out. The interviewer wants a systematic hunter, not a lucky guesser, so make the reasoning visible. End with the root cause and the fix, and, ideally, the process change that prevents its recurrence.

Q2. *Describe a time you disagreed with a teammate or a reviewer.*
A. Assessing: whether you can hold a technical position and also collaborate. Choose a disagreement that was resolved *constructively* and where you were at least partly persuaded or persuading on the merits—not one where you simply won or capitulated. Show that you argued from data (timing numbers, a coverage gap, an interface risk), listened to the other view, and reached a decision the team could commit to. Avoid stories that cast the other person as incompetent; they signal that you will be hard to work with.

Q3. *Tell me about a project you are proud of.*
A. Assessing: the scope of what you can own and how you talk about your own contribution. Pick something with real technical substance and be scrupulous about *your* role versus the team's—inflating your contribution is a serious red flag, and interviewers probe for it. Explain a hard decision you made and why, and give the outcome. This is your best chance to steer the conversation toward your strengths, so choose a project that showcases the skills the role needs.

Q4. *Tell me about a mistake you made and what you learned.*
A. Assessing: self-awareness and growth, not confession. Choose a real, non-catastrophic mistake—a missed corner case, an assumption that did not hold, a review you rushed—own it plainly without blaming others, and spend most of the answer on what you changed as a result. "I now always add a CDC review gate before sign-off" turns a mistake into evidence of judgment. Never claim you have never made one.

Q5. *How do you approach an ambiguous or underspecified requirement?*
A. Assessing: whether you flail or drive clarity. Describe your process: identify the specific unknowns, list the assumptions you would have to make, take them back to the stakeholder or architect with proposed answers rather than open questions, and document the decision. Hardware amplifies the cost of building the wrong thing, so showing that you resolve ambiguity *before* committing RTL is exactly the maturity a senior role wants.

Q6. *Why this company or this role?*
A. Assessing: whether you have thought about fit or are carpet-bombing. Connect something specific about their product or technology to your interests and experience. A vague "you're a great company" answers nothing; "you build high-speed data-path FPGAs and I've spent two years on exactly that kind of CDC and timing-closure work" shows you did your homework.

C. Questions to Ask Them

Every interview ends with "do you have questions for us," and having none reads as disinterest. Prepare a few that are genuine and reveal how the team works: what the verification and sign-off flow looks like, how design reviews are run, what the hardest current technical problem is, how success in the role is measured, and what the path from RTL to silicon or bitstream looks like on their projects. Good questions double as a signal that you think about process and quality, not just writing code.

D. Calibrating to Level

Match the scope of your stories to the level you are interviewing for (§I). A junior candidate's stories can center on a well-executed module and a bug hunted down under guidance. A mid-level candidate should show ownership of a subsystem and unprompted attention to timing, CDC, and verification. A senior candidate is expected to have driven cross-cutting decisions—a clocking architecture, an interface choice, a verification strategy—and to talk about mentoring, tradeoffs, and outcomes at the level of a whole design rather than a block. The same episode can often be told at any of these altitudes; choose the altitude that fits the role.

XX. A Study Plan and Further Reading

Knowing the material and being able to *perform* it under interview pressure are different skills, and only the second wins offers. This closing section turns the guide into a plan: what to study in what order, how to practise so the knowledge is retrievable on demand, and where to go deeper.

A. A Two-Week Plan

Two weeks of focused evening study is enough to cover this guide with time to rehearse, provided you front-load the highest-yield material. The plan in Table IV does exactly that: it puts static timing (§V) and metastability/CDC (§VI)—the two topics the introduction singled out as non-negotiable (§I)—on the first two days, while you are freshest, and returns to them at the end for rehearsal. Scale it to the time you have: if you have one week, do two topics a day and skip the second pass; if you have a month, add a mock interview every few days.

B. How to Practise

Reading is necessary but not sufficient; the interview is a performance, so practise performing.

- **Answer out loud.** Every question in this guide should be answered spoken, as if to an interviewer, not silently. The gap between "I know this" and "I can say this clearly in ninety seconds" is exactly what the first pass reveals, and it is wide for the timing and CDC explanations in particular.
- **Write RTL from a blank editor.** Re-derive the idioms of §XI and the whiteboard solutions of §XVII without looking—an edge detector, a synchronizer, a small FSM, a synchronous FIFO. On a real whiteboard there is no autocomplete and no simulator; fluency comes only from having written them before.
- **Do timed mock interviews.** Have someone ask you questions cold, or set a timer and talk to a camera. The pressure of a clock and an audience surfaces the answers that are not yet automatic.
- **Explain, don't just answer.** Practise narrating your reasoning as you work, because that is what is graded (§I). A correct answer delivered silently scores worse than a well-reasoned near-miss.

C. A Priority Checklist

If time is short, master these in order; each must be automatic before you move on. (1) Explain setup and hold, and why slowing the clock fixes one but not the other (§V). (2) Explain metastability, the two-flop synchronizer, and multi-bit crossing (§VI). (3) Write a synchronizer, an edge detector, and a small FSM from memory (§XI, §IV). (4) State the blocking/nonblocking rule and what infers a latch (§IX, §X). (5) Describe a LUT, block RAM, and the FPGA resource choices (§VIII). (6) Design a synchronous FIFO and compute a FIFO depth (§XII). (7) Explain pipelining's latency / throughput / $F_{\max}$ tradeoff (§XIII). (8) Have three STAR stories ready (§XIX).

D. Further Reading

For the fundamentals and architecture, Harris and Harris [1] is the standard modern text and covers digital logic through FPGA and processor design at exactly interview depth. For HDL semantics and the pitfalls that generate spot-the-bug questions, Cummings's SNUG papers are the canonical sources—nonblocking assignments and coding styles [5], and clock-domain crossing [3]—and Chu [7] is a thorough RTL text for the VHDL side. Ginosar's tutorial [6] is the reference for the metastability mathematics if an interviewer pushes on MTBF. Vendor methodology guides [10], [11] ground the timing-closure and synthesis questions in real tool behavior.

For candidates who want to go past interview depth into how the pieces are actually built, the companion *RTL Processor Design* series [14] maps onto this guide topic by topic: its Volume II treats FPGA implementation, timing, clocking, and pipelining at length; Volume V is a full deep-dive on clock-domain crossing—the single highest-yield interview topic—going well beyond §VI; and Volumes I, III, and IV cover processor microarchitecture, logic synthesis, and physical implementation for the system-level and internals questions a senior interview may reach.

E. The Night Before

Do not learn anything new the night before; consolidate. Skim the rapid-fire bank (§XVIII) once, re-read your two timing and CDC explanations until they are automatic, review your STAR stories, sleep. Walk in prepared to think out loud, to ask clarifying questions before you design, and to reason through the one question you did not anticipate—because there will be one, and handling it with visible, structured thinking is what separates a strong candidate from a merely well-drilled one. Good luck.

References

[1] S. L. Harris and D. M. Harris, *Digital Design and Computer Architecture: RISC-V Edition*. Morgan Kaufmann, 2021.

[2] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Addison-Wesley, 2010.

[3] C. E. Cummings, "Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog," in *SNUG Boston*, 2008.

[4] C. E. Cummings, "Simulation and Synthesis Techniques for Asynchronous FIFO Design," in *SNUG San Jose*, 2002.

[5] C. E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!," in *SNUG Boston*, 2000.

[6] R. Ginosar, "Metastability and Synchronizers: A Tutorial," *IEEE Design & Test of Computers*, vol. 28, no. 5, pp. 23–35, 2011.

[7] P. P. Chu, *RTL Hardware Design Using VHDL*. Wiley-IEEE, 2006.

[8] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd ed. Prentice Hall, 2003.

[9] S. Sutherland and D. Mills, "Standard Gotchas: Subtleties in the Verilog and SystemVerilog Standards," in *SNUG*, 2006.

[10] AMD/Xilinx, "UltraFast Design Methodology Guide (UG949)," 2023.

[11] AMD/Xilinx, "Vivado Design Suite User Guide: Synthesis (UG901)," 2023.

[12] IEEE, "IEEE Standard for SystemVerilog (IEEE 1800-2017)," 2018.

[13] IEEE, "IEEE Standard VHDL Language Reference Manual (IEEE 1076-2019)," 2019.

[14] RTL Processor Design Series, "Volumes I–V" (RISC-V core; FPGA implementation; logic synthesis; physical implementation; clock-domain crossing), 2026.

TABLE IV
A two-week evening study plan, front-loading the highest-yield topics. Adjust the pace to the time available; the ordering is the important part.

Day	**Focus**	**Sections**
1	Static timing analysis (rehearse aloud)	§V
2	Metastability, synchronizers, CDC	§VI
3	Clocking and reset	§VII
4	Sequential logic and FSMs	§IV
5	FPGA architecture	§VIII
6	Verilog/SystemVerilog semantics and pitfalls	§IX
7	VHDL semantics and pitfalls	§X
8	RTL coding idioms (write each from scratch)	§XI
9	Memory and FIFO design	§XII
10	Pipelining; interfaces and handshakes	§XIII, §XIV
11	Verification; debug and closure	§XV, §XVI
12	Whiteboard problems (timed, on paper)	§XVII
13	Fundamentals + arithmetic; rapid-fire bank	§II, §III, §XVIII
14	Rehearse timing + CDC aloud; behavioral stories	§V, §VI, §XIX